

ARBEITSBERICHTE DES INSTITUTS FÜR MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG (INFORMATIK)

FRIEDRICH ALEXANDER UNIVERSITÄT ERLANGEN NÜRNBERG

HERAUSGEBER: H. Billing, A. Bode, W. Händler, U. Herzog,
F. Hofmann, K. Leeb, P. Mertens, H. Müller, G. Nees, H. Niemann,
B. Schmidt, H. J. Schneider, D. Seitzer, H. Wedekind

EINE PROGRAMMIERUMGEBUNG FÜR VERTEILTES PEARL

CODEERZEUGUNG MITTELS PROGRAMMSYNTHESE

Gudrun Kragl

DISSERTATION

BAND 19 · NUMMER 8 · ERLANGEN · OKTOBER 1986

1986 № 452

ISSN 0344 - 3515

**EINE PROGRAMMIERUMGEBUNG
FÜR VERTEILTES PEARL
CODEERZEUGUNG MITTELS PROGRAMMSYNTHESE**

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades
DOKTOR - INGENIEUR

vorgelegt
von

Gudrun Kragl

ERLANGEN 1986

Als Dissertation genehmigt von der
Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 11.7.1986

Tag der Promotion: 16.9.1986

Dekan: Prof. Dr.-Ing. H. Brand

Berichterstatter: Prof. Dr. H.-J. Schneider

Prof. Dr. F. Hofmann

Kurzfassung

Die vorliegende Arbeit beschreibt eine Programmierumgebung für verteiltes PEARL. Verteiltes PEARL umfaßt im wesentlichen Avionic-PEARL und Sprachkonstrukte zur zeitüberwachten Kommunikation von Prozessen über Botschaften sowie nichtdeterministische Kontrollanweisungen.

Der Anwender der Programmierumgebung wird in der Spezifikationsphase und während der Programmierung unterstützt.

Für die graphischen Teile der Spezifikationsmethode PASS, d.h. für Kommunikationsstruktur und Ablaufsteuerung, steht ein Zeichenprogramm zur Verfügung. Durch die nichtgraphischen Teile der Spezifikationsmethode wird der Benutzer mit Masken geführt. Mittels Programmsynthese wird aus der Spezifikation automatisch PEARL-Code erzeugt.

Dazu wird eine um semantische Aktionen erweiterte PEARL-Grammatik nach dem Verfahren des rekursiven Abstiegs programmiert. Der Syntheseweg wird durch Informationen aus der Spezifikation und fest definierte Umsetzregeln gesteuert.

Zur Unterstützung der Programmierung im Großen wird die in der Spezifikation beschriebene Zerlegung auf PEARL-Moduln abgebildet. Auf Anweisungsebene, bei der Programmierung im Kleinen, orientiert sich die Umsetzung am graphischen Objekt Knoten aus der Ablaufsteuerung.

Das Ergebnis der Programmsynthese ist ein PEARL-Programngerüst.

Inhaltsverzeichnis

1.	Einleitung - Motivation	1
2.	Software-Entwicklungsumgebungen	4
2.1.	Anforderungen an eine Software-Entwicklungs- umgebung	4
2.2.	EPOS	8
2.2.1.	Grundsätzlicher Aufbau des EPOS-Systems	9
2.2.2.	Die Spezifikationssprache EPOS-R	11
2.2.3.	Die Spezifikationssprache EPOS-S	17
2.2.4.	Die Spezifikationssprache EPOS-P und das Manage- ment-Programmsystem EPOS-M	25
2.2.5.	Die Bedienung des EPOS-Arbeitsplatzrechners mit EPOS-C	27
2.2.6.	Der Übergang von der EPOS-S Spezifikation zum Programmcode	27
2.3.	Software-Entwicklung mit Hilfe eines Transforma- tions-Expertensystems	36
2.3.1.	Das Transformationssystem TREX	36
2.3.2.	Anwendung von TREX für Prozeßsteuerungssoftware	39
3.	Automatisches Programmieren	40
3.1.	Begriffsklärung und Motivation	40
3.2.	Möglichkeiten zur Computerunterstützung der Pro- grammierung	41
3.3.	Programmtransformation und Programmsynthese	43
3.3.1.	Programmtransformation	43
3.3.2.	Programmsynthese	44
4.	Die Spezifikationsmethode PASS	53
4.1.	Beschreibung der Kommunikationsstruktur	53
4.2.	Beschreibung der Definition des Prozeßsystems	54
4.2.1.	Die Kommunikationsmaschine	55
4.2.2.	Die Ablaufsteuerung	55
4.2.3.	Die private Benutzermaschine	57
4.2.4.	Die gemeinsame Benutzermaschine	57
4.3.	Gliederungsschema der Spezifikation	57

5.	Eine Programmierungsumgebung für verteiltes PEARL	59
5.1.	Einordnung in den Software-Lebenszyklus	59
5.2.	Benutzersicht	60
5.3.	Interne Sicht: Überblick über die Programmteile, Datenstrukturen und Schnittstellen	65
6.	Datenstrukturen	69
6.1.	Datenstrukturen der Spezifikationsphase	69
6.1.1.	Listen	69
6.1.2.	Datenstruktur Masken	73
6.1.3.	Datenstruktur Prozeßsystem	74
6.2.	Synthese-Phase	76
6.2.1.	Umsetzregeln	76
6.2.2.	PEARL-Programmbaum	78
7.	Schnittstellenbeschreibung	80
7.1.	Die Schnittstelle zwischen den graphischen und nichtgraphischen Teilen der Spezifikations- methode	80
7.2.	Die Schnittstelle zwischen Spezifikation und Synthese	81
8.	Beschreibung der Programmteile	87
8.1.	Programmierung der erweiterten PEARL-Grammatik	87
8.2.	Maskensteuerprogramm	97
9.	Ausblick	99

Anhang 1: Erweiterte PEARL-Grammatik

Anhang 2: Vordefinierte Symbole zum Erstellen der
 Kommunikationsstruktur und der Ablaufsteuerung

Anhang 3: Masken

1. Einleitung - Motivation

Ziel dieser Arbeit ist die Entwicklung einer Programmierumgebung für verteiltes PEARL. Verteiltes PEARL enthält Sprachkonstrukte zur Kommunikation von Prozessen über Botschaften und nichtdeterministische Kontrollanweisungen. Die Sprache wird hauptsächlich in der verteilten Prozeßsteuerung und zur Implementierung von Kommunikationsnetzen eingesetzt.

Aus diesen Anwendungsgebieten erklären sich die im nachfolgenden aufgeführten Anforderungen an die Programmierumgebung für verteiltes PEARL.

Bereits bei der Spezifikation eines Prozeßsystems soll der Anwender durch einen Arbeitsplatzrechner unterstützt werden. Als Spezifikationsmethode wird die graphisch orientierte Methode PASS vorausgesetzt. Zum Erstellen der PASS-Graphiken steht dem Benutzer ein leicht bedienbares, menüorientiertes Zeichenprogramm zur Verfügung. Als weitere Unterstützung soll der Benutzer durch die nichtgraphischen Teile der Methode mit Masken und Menüs geführt werden.

Überprüfungen auf Vollständigkeit und Konsistenz während der Spezifikation und eine möglichst automatische Umsetzung der Spezifikation in ein korrektes PEARL-Programmgerüst sollen die Zuverlässigkeit der zu erstellenden Prozeßsoftware erhöhen.

Durch die geplante automatische Umsetzung in Code (Programmsynthese) soll erreicht werden, daß sich die Entwickler weniger mit der Programmierung als mit der Spezifikation beschäftigen. Darüberhinaus soll der für die Programmentwicklung benötigte Zeitaufwand reduziert werden.

Weitere Anforderungen sind eine Verbesserung der Dokumentation durch Ausgabe der in der Spezifikationsphase gesammelten Informationen. Die Dokumentation kann nach jeder Änderung in der Spezifikation auf Anforderung ausgegeben werden.

Die einheitliche Struktur der automatisch erzeugten PEARL-Programmgerüste mit den darin enthaltenen Kommentaren soll leichteres Ändern und Ergänzen und späteres Warten der Programme ermöglichen.

Die Programmierumgebung wird modular aufgebaut, so daß vom Anwender gegebenenfalls nur die Werkzeuge zur Unterstützung

der Spezifikation (ohne Programmsynthese) verwendet werden können. Dies ist denkbar, wenn als Zielsprache nicht PEARL gewünscht wird.

Bei der Umsetzung nach PEARL wird sowohl die Programmierung im Großen, d.h. die Bildung einer Modulstruktur, als auch das Programmieren im Kleinen auf Anweisungsebene unterstützt. Bei der Synthese von PEARL-Programmen sind drei Aspekte von Bedeutung. Zum einen wird zur Codeerzeugung Information aus den in der Spezifikation erzeugten graphischen Objekten verwendet. Zum anderen liegt der Schwerpunkt bei der Synthese auf Sprachkonstrukten zur zeitüberwachten Kommunikation von Prozessen über Botschaften; d.h. auf Sprachkonstrukten, die in verteilten Systemen benötigt werden. Schließlich zeigt die Wahl der Zielsprache PEARL, daß auch die Erzeugung von Anweisungen zur Kommunikation mit dem technischen Prozeß unterstützt werden soll.

Allgemeine Anforderungen an Software-Entwicklungsumgebungen, deren Bestandteile und Werkzeuge werden in Kapitel 2 beschrieben. Als Beispiel einer existierenden Software-Entwicklungsumgebung wird EPOS ausführlich vorgestellt. TREX soll die neueren Ansätze in Software-Entwicklungsumgebungen aufzeigen.

Kapitel 3 beginnt mit einer Definition des Begriffs Automatisches Programmieren. In dieser Arbeit sind unter Automatischem Programmieren Methoden und Werkzeuge zur Computerunterstützung des Programmiervorgangs zu verstehen. Die grundsätzlichen Wege zur Computerunterstützung des Programmiervorgangs werden aufgezeigt. Es schließen sich Beispiele zu Programmtransformation und Programmsynthese an.

Die restlichen Kapitel enthalten eine Beschreibung der Programmierungsumgebung für verteiltes PEARL.

In Kapitel 4 werden die wichtigsten Bestandteile der Spezifikationsmethode PASS, soweit sie zum Verständnis der nachfolgenden Kapitel benötigt werden, beschrieben.

Kapitel 5 bringt einen Überblick über die Programmierungsumgebung für verteiltes PEARL.

Kapitel 6 bis 8 behandeln ausführlich die in der Programmierungsumgebung enthaltenen Datenstrukturen, Schnittstellen und Programmteile.

Kapitel 9 geht auf die eingangs formulierten Anforderungen ein und zeigt, inwieweit diese von der Programmierumgebung erfüllt werden. Die Arbeit schließt mit einer Kritik am bestehenden Ansatz und einem Ausblick auf weiterführende Arbeiten.

2. Software-Entwicklungsumgebungen

Im Kapitel 1 wurden spezielle Anforderungen an die Programmierungsumgebung für verteiltes PEARL aufgestellt. In diesem Kapitel wird zunächst der Begriff Software-Entwicklungsumgebung geklärt. Anschließend wird erläutert, welche Anforderungen an die Software-Entwicklungsumgebung allgemein gestellt werden, d.h. welche Motivation bei der Entwicklung vorliegt, welche Ziele verfolgt werden und welche Werkzeuge eine Software-Entwicklungsumgebung enthalten kann.

Als Beispiele für Software-Entwicklungsumgebungen werden EPOS und TREX genauer beschrieben.

2.1. Anforderungen an eine Software-Entwicklungsumgebung

Die nachfolgenden Ausführungen lehnen sich an den in /HÖNK81/ formulierten Fragenkatalog zur Analyse einer Software-Entwicklungsumgebung und an den von /HAUS81/ erstellten Überblick über den Entwicklungsstand und Trends von Software-Produktions-Umgebungen an.

In /HAUS81/ wird der Begriff Software-Entwicklungsumgebungen (SEU) oder Software-Produktionsumgebungen (SPU) folgendermaßen charakterisiert:

"Der Begriff Software-Produktions-Umgebung bezeichnet ein instrumentiertes und organisiertes Software-Entwicklungs-Laboratorium, in dem viele Personen arbeiten, um gemeinsam in einem vollständig organisierten Arbeitsprozeß Software zu entwerfen, zu konstruieren, zu prüfen, zu ändern und zu warten. Eine SPU bietet software-gestützte Modelle, Methoden, Verfahren, Beschreibungsmittel und Werkzeuge für diese Arbeit. SPUs unterstützen die Software-Entwicklung und -Anwendung dadurch, daß sie diese Mittel bereitstellen und dadurch, daß sie die Handhabung dieser Mittel festlegen."

Einfacher ausgedrückt, Software-Entwicklungsumgebungen unterstützen ein Entwicklungsteam beim Erstellen von Software in allen Phasen des Software-Lebenszyklus.

Ziele, die bei der Entwicklung von Software-Entwicklungsumgebungen angegeben werden, sind Kostenreduzierung, Produktivitätssteigerung, Standardisierung, Normung und Verbesserung der Softwarequalität.

Je nachdem, ob es sich um Software-Produktionsumgebungen in der Industrie, im Forschungsbereich oder in öffentlichen Bereichen handelt, stehen andere Ziele im Vordergrund. Die Motivation für die Entwicklung und Anwendung einer SPU in der Industrie liegt letztlich in der Bewältigung großer Produktionsprojekte. Wirtschaftliche und marktstrategische Ziele sind ausschlaggebend. Forschung interessiert nur dann, wenn sie zur Lösung der eigenen, gerade aktuellen Probleme beiträgt.

Standardisierung und Normung von Produkten und Produktionsverfahren stehen bei der Entwicklung und beim Einsatz von Software-Entwicklungsumgebungen im öffentlichen Bereich im Vordergrund.

Software-Entwicklungsumgebungen bieten Hilfsmittel und Werkzeuge für die verschiedenen Phasen des Software-Lebenszyklus an. Dies können "einfache" Hilfsmittel zur Programmierung (Editor, Compiler, Binder, Lader) oder komplexe Werkzeuge zur Unterstützung der Anforderungsanalyse, der Spezifikation, des Systementwurfs, der Produktkontrolle und des Softwaremanagements sein. Rechnerunterstützung wird nicht nur für alle Phasen, sondern auch für alle Tätigkeitsbereiche während der Entwicklung eines Projekts gefordert.

Von integrierten Software-Entwicklungsumgebungen wird gesprochen, wenn nicht für jedes einzelne Werkzeug eine eigene Benutzerschnittstelle mit einer eigenen Kommandosprache vorliegt, sondern wenn für alle Werkzeuge eine einheitliche Benutzerschnittstelle existiert.

Folgende Komponenten werden von vielen SPU-Entwicklern als wesentlich angesehen:

- Informationssysteme für technische Mitarbeiter und für das Management (Software-Datenbanken, Projektbibliothek)
- Sprachen für die Anforderungsdefinition (sowohl technische Anforderungen als auch Managementanforderungen) sowie für Systemspezifikation und -Implementierung, instrumentiert

- mit Sprachumgebungen (Compiler, Editor,...)
- Werkzeuge zur graphischen Darstellung von Software
 - Instrumente und Verfahren zur Kontrolle und Verbesserung der Software-Qualität
 - Arbeitsplätze für technische Mitarbeiter und Manager sowie
 - Werkzeuge für die Projektabwicklung, d.h. Projektplanung, Projektführung und Projektkontrolle

In einer Software-Datenbank werden die verschiedenen Beschreibungen von Software, d.h. die Grob- und Feinspezifikationen und Programmtexte, eventuell in mehreren Versionen und Varianten, gespeichert. Damit werden die Aufgaben der technischen Produktion unterstützt. Ohne Datenbanken könnte die Fülle an Information, die für die verschiedenen Phasen des Software-Lebenszyklus vorliegt, nicht mehr überschaubar und konsistent gehalten werden.

Von Projektbibliotheken wird gesprochen, wenn neben der Produktverwaltung auch die Verwaltung der Management-Informationen unterstützt wird.

Viele Software-Produktionsumgebungen bieten Werkzeuge für das Programmieren in einer bestimmten Programmiersprache an. Sprachen und die dazugehörigen Instrumente wie Editor, Compiler oder Interpreter, Binder, Lader und Laufzeitsystem sind wesentliche Elemente einer SPU. Es existieren auch sprachunabhängige Software-Produktionsumgebungen, oder SPUs, die mit mehr oder weniger Aufwand auf eine neue Zielsprache umgestellt werden können. Die Werkzeuge für die verschiedenen Sprachen sollten soweit als möglich gemeinsame Teile verwenden und sich nur an den sprachspezifischen Stellen unterscheiden.

Entwicklungsumgebungen, die überwiegend die Codierung unterstützen, werden als Programmierungsumgebungen bezeichnet. Sie enthalten als Werkzeuge z.B. Editor und Compiler oder Interpreter.

Software-Entwicklungsumgebungen, die durch Erweiterungen von Programmierungsumgebungen entstanden sind, enthalten neben den eben beschriebenen Sprachen und Werkzeugen auch Sprachen zur Anforderungsdefinition und/ oder zur Systemspezifikation.

Graphiken eignen sich besonders, um komplexe Strukturen dar-

zustellen. Einige SPUen bieten deshalb gerade für die frühen Phasen des Software-Entwurfs rechnergestützte Werkzeuge für die Ausgabe von Diagrammen etc. In den meisten Fällen können nur textuelle Eingaben graphisch dargestellt werden, selten werden Daten aus graphischen Beschreibungen weiter genutzt. Textuelle und graphische Beschreibungsmittel können meistens nicht in einem Arbeitsschritt eingesetzt werden.

Software-Produktionsumgebungen sollten auch Werkzeuge für Qualitätssicherung, Abnahmetest und Verifikation bzw. Validation enthalten.

Qualitätssicherung bedeutet dabei Definition und Kontrolle von Software-Standards. Bei einem Abnahmetest wird dem Kunden oder Benutzer vorgeführt, daß die Software die an sie gestellten Anforderungen erfüllt, d.h. daß die Software das tut, was sie tun soll. Verifikation und Validation bedeuten die Überprüfung von Software auf Fehlerfreiheit. Dabei wird Validation als Test verstanden, Verifikation als formale Prüfung.

Als Werkzeuge stehen in einigen Software-Produktionsumgebungen Trace- oder Schritt-orientierte Testsysteme oder Debugger zur Verfügung.

Komplexere Werkzeuge bestehen aus Instrumenten und Verfahren, um Tests zu planen, zu spezifizieren und Ergebnisse zu protokollieren.

Zur Unterstützung des Managements, d.h. für die Planung, Kontrolle und Steuerung von Software-Produktion und deren Anwendung, können die Werkzeuge Projektbibliothek und Management-Informationssysteme eingesetzt werden.

Projektführungssysteme, die die Projektplanung unterstützen, betrachten die Managementanforderungen, d.h. die nichttechnischen Daten wie Grenzkosten, Projektendetermin, Richtlinien und Vorüberlegungen zur Projektplanung, zur Projektorganisation und Überwachung, meistens getrennt von den technischen Anforderungen.

Um eine genauere Vorstellung von diesen Werkzeugen zu erhalten, wird im folgenden die Software-Entwicklungsumgebung EPOS ausführlich behandelt.

2.2. EPOS

"EPOS (Entwicklungsunterstützendes Prozeß-Orientiertes Spezifikationssystem) ist ein Werkzeug für Ingenieure, die informationsverarbeitende technische Systeme projektieren, entwickeln und warten. Es ermöglicht den Aufbau rechnerunterstützter Arbeitsplätze für die Ingenieur Tätigkeiten während des gesamten Lebenslaufs eines Projekts. Gleichzeitig stellt EPOS ein wirksames Hilfsmittel für die Software-Qualitätssicherung und für das Projektmanagement dar." /EPOS83b/

EPOS soll, wie der Name andeutet, die Entwicklung von Realzeitsystemen, besonders bei der Prozeßautomatisierung, unterstützen.

EPOS-Arbeitsplatzrechner stehen allen an einem Projekt beteiligten Ingenieuren (Projektierern, Programmierern, Elektronikern, Projektleitern) zur Verfügung. EPOS ermöglicht damit auch eine verbesserte Kommunikation zwischen den Projektbeteiligten.

Entwickelt wurde EPOS am Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart. Praktisch eingesetzt wird EPOS seit 1980 bei Anwendern in Industrie, Forschung und bei Behörden. EPOS steht z.B. auf den Rechnern Siemens 7000 oder DEC VAX zur Verfügung. Implementierungen auf leistungsfähigen Mikrorechnern sind in Vorbereitung.

Die Abbildungen in diesem Kapitel wurden aus /EPOS83a/ entnommen.

2.2.1. Grundsätzlicher Aufbau des EPOS-Systems

Bild 1 zeigt, wie bei der Anwendung von EPOS grundsätzlich vorzugehen ist.

Mit Hilfe von Spezifikationssprachen (Beschreibungssprachen) werden die während der Entwicklung erarbeiteten Informationen (d.h. Zielsetzung, Aufgabenstellung, Anforderungen, Informationen für das Management ...) formuliert und in den EPOS-Arbeitsplatzrechner eingegeben. Aus diesen aktuellen Informationen wird eine Datenbank aufgebaut. Die Informationen in der Datenbank können z.B. auf Vollständigkeit, Konsistenz und Widerspruchsfreiheit überprüft werden. Auf Anforderung des Benutzers können von Dokumentationsprogrammen generierte Texte und Graphiken, Soll-Ist-Vergleiche der Termin- und Kostensituation oder Ergebnisse sonstiger Prüfprogramme ausgegeben werden.

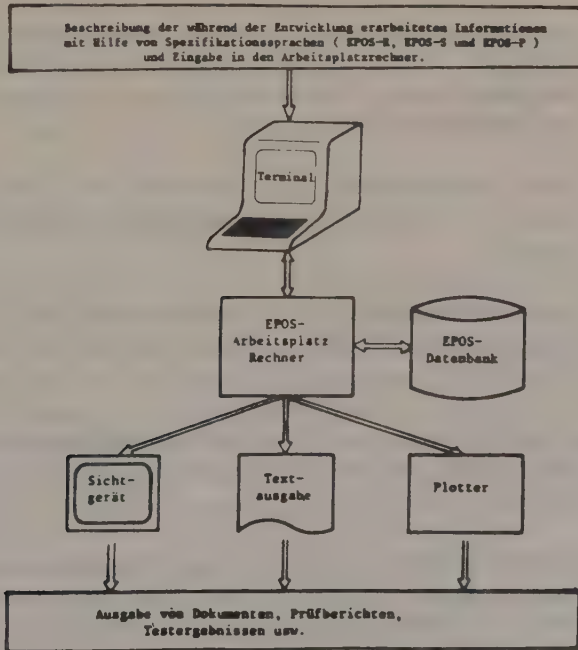


Bild 1: Vorgehensweise bei der Anwendung von EPOS

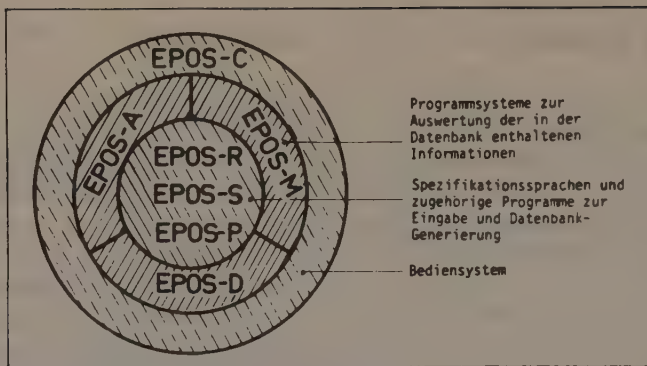


Bild 2: Die Komponenten von EPOS

Bild 2 verdeutlicht den Aufbau eines EPOS-Systems. Die EPOS-Komponenten lassen sich folgendermaßen kurz charakterisieren:

EPOS-R Spezifikationssprache für die frühen Phasen eines Projekts. Sie wird eingesetzt zur Beschreibung der Aufgabenstellung (Pflichtenheft) und zur Beschreibung des **konzeptionellen** Entwurfs.

EPOS-S Spezifikationssprache mit formaler Syntax und definierter Semantik zur Beschreibung des **operationellen** Entwurfs.

EPOS-P Spezifikationssprache zur Beschreibung von Informationen für das Projektmanagement und die Produktverwaltung.

EPOS-M Programmsystem zur Unterstützung für die Management-Tätigkeiten bei der Projektführung und Produktverwaltung.

EPOS-A Programmsystem zur Analyse und Überprüfung der mit EPOS-R, EPOS-S und EPOS-P beschriebenen Sachverhalte, sowie zur Erstellung von Prüf- und Ergebnisberichten.

EPOS-D Programmsystem zur Erstellung von Dokumenten aus den mit EPOS-R, EPOS-S und EPOS-P beschriebenen Informa-

tionen.

EPOS-C Bediensystem zur Kommunikation zwischen Benutzern und EPOS-Arbeitsplatzrechner.

Konzeptioneller Entwurf beinhaltet dabei die Analyse des technischen Prozesses, das Aufstellen von Modellen und das Suchen und Finden einer Lösungskonzeption einschließlich entsprechender Teillösungen, Lösungskomponenten, Lösungsalgorithmen etc.

Operationeller Entwurf heißt, mit Hilfe der gewählten Lösungskonzeption ein technisches System (Automatisierungssystem) zu entwerfen. Dieses technische System soll den Betrieb des technischen Prozesses gemäß der Aufgabenstellung gewährleisten.

Als dritte Entwurfsart gibt es in EPOS noch den "auxiliären" Entwurf. Dabei werden Hilfsmittel wie Testgeräte, Testprogramme, Wartungseinrichtungen oder Simulationsprogramme definiert.

Die Bestandteile und die Arbeitsweise der EPOS-Komponenten werden in den folgenden Kapiteln genauer vorgestellt.

2.2.2. Die Spezifikationssprache EPOS-R

Die Spezifikationssprache EPOS-R wird zur Beschreibung der Aufgabenstellung (Pflichtenheft) und des konzeptionellen Entwurfs eingesetzt. Da bei der Festlegung der Aufgabenstellung und des Entwurfs viele Personen mit unterschiedlicher Ausbildung beteiligt sind, wie z.B. kaufmännische Planer, Prozeßfachleute, Software-Entwickler und Gerätehersteller, enthält EPOS-R wenige formale Sprachmittel. Stattdessen läßt EPOS-R eine weitgehend informelle verbale Beschreibung zu.

Die formalisierten, leicht verständlichen Beschreibungsmittel sind:

- ein **einheitliches Gliederungsschema** für die Aufgabenstellung und den konzeptionellen Entwurf
- **Entscheidungsstabellen** zur formalen Beschreibung von Entscheidungs-Zusammenhängen

- ein **Lexikon** zur Definition, Auskunft und Vervollständigung von Begriffen, die in den natürlich-sprachlichen Texten verwendet werden
- **Identifizierbare Aufgabenkomponenten**, d.h. formale Einschübe; sie erlauben formale Prüfungen und stellen den Zusammenhang zu EPOS-S her.

Bild 3 bringt einen Ausschnitt aus dem Gliederungsschema für die Aufgabenstellung.

1. Zielsetzung
 - 1.1 Beschreibung der wirtschaftlichen, soziologischen, ökologischen und politischen Zusammenhänge ("Umgebendes System").
 - 1.2 Zielsetzung des Prozeßautomatisierungssystems
 - 1.3 Rolle des Menschen
2. Voraussetzungen
 - 2.1 Voraussetzungen bezüglich des Vorgehens bei der Klärung der Aufgabenstellung und der Anforderungen.
 - 2.2 Voraussetzungen bezüglich der Einbettung des Prozeßautomatisierungssystems in das "Übergeordnete System" (Ebene-0-System).
3. Ist-Zustand
 - 3.1 Ist-Zustand des technischen Prozesses
 - 3.1.1 Statische Beschreibung
 - 3.1.2 Dynamische Beschreibung (regulärer Betrieb)
 - 3.1.3 Beschreibung irregulärer Prozeßzustände und -abläufe
 - 3.2 Ist-Zustand eines bisher vorhandenen Automatisierungssystems
 - 3.2.1 Bisherige Aufgaben des vorhandenen Automatisierungssystems
 - 3.2.2 Vorhandene Geräte (z.B. Regelgeräte, Bedienkonsole usw.)
 - 3.2.3 Vorhandene Rechner und zugehörige Peripherie
 - 3.2.4 Vorhandene Software (z.B. Entwicklungssoftware, Betriebssystem usw.)
4. Schnittstellen
 - 4.1 Schnittstellen zum technischen Prozeß
 - 4.2 Schnittstellen zum Bedienpersonal
 - 4.3 Schnittstellen zu anderen Teilsystemen des "Übergeordneten Systems"
5. Funktionelle Anforderungen im regulären Betrieb
 - 5.1 Auflistung der Automatisierungs-Aufgabenstellung im regulären Betrieb
 - 5.2 Beschreibung geforderter Reaktionen auf die möglichen Eingabedaten
 - 5.3 Forderung bezüglich der Genauigkeit der Ausgabedaten
 - 5.4 Zeitliche Anforderungen des regulären Betriebes
6. Beforderte Reaktionen auf unerwünschte Ereignisse (irregulärer Betrieb)
 - 6.1 Hardwarefehler
 - 6.2 Falsche Eingabedaten
 - 6.3 Falsche interne Daten

Bild 3: Gliederungsschema für die Aufgabenstellung

Dieses Gliederungsschema umfaßt Kapitel, die in Sektionen untergliedert werden können. Kapitel und Sektionen sind jeweils mit einer frei wählbaren Überschrift zu versehen.

Bild 3 enthält z.B. Kapitel 4 mit der Überschrift 'Schnittstellen' und die in Kapitel 4 enthaltene Sektion 4.1 mit der Überschrift 'Schnittstellen zum technischen Prozeß'.

Kapitel und Sektionen können aus natürlich-sprachlichen Texten, formalen Einschüben, Begriffen aus dem Begriffslexikon, Quittierungen und Ersetzungen von Aufgabenkomponenten aufgebaut sein.

Obiges Gliederungsschema kann bei der Formulierung der Ziele, Aufgabenstellung und Anforderungen als Checkliste verwendet werden. Angepaßt an das jeweilige Projekt, kann das Schema ergänzt oder verändert werden.

Ein analoges Gliederungsschema existiert für die Beschreibung des konzeptionellen Entwurfs.

Entscheidungstabellen sind, obwohl ein formales Beschreibungsmittel, leicht verständlich.

Da Entscheidungstabellen auch noch in einem späteren Zusammenhang benötigt werden, soll der grundsätzliche Aufbau einer Entscheidungstabelle anhand von Bild 4 erläutert werden.

Identifikationsteil				
Bedingungsliste	Regel 1	Regel 2	...	ELSE
Bedingung 1				
Bedingung 2				
⋮				
Bedingung m				
Massnahmenliste:				
Massnahme 1				
Massnahme 2				
⋮				
Massnahme n				

Bedingungs-
anzeiger

Massnahmen-
anzeiger

Bild 4: Grundsätzlicher Aufbau einer Entscheidungstabelle

Eine Entscheidungstabelle wird in vier Teilfelder gegliedert. Der Identifikationsteil enthält einen frei wählbaren Bezeichner.

In der Bedingungsliste werden die zu verknüpfenden Bedingungen aufgeführt. (CONDITIONLIST)

In der Maßnahmenliste sind alle auszuführenden Aktionen bzw. Maßnahmen zusammengestellt. (OPERATIONLIST)

Der Regelteil enthält die einzelnen Entscheidungsregeln. (RULES)

Im Bedingungsanzeiger werden die Werte der Bedingungen dargestellt, z.B. das Erfülltsein oder Nicht-Erfülltsein einer Bedingung (Ja oder Nein in der Tabelle). Ein '-' bedeutet, daß es in der Entscheidungsregel keine Rolle spielt, ob die betreffende Bedingung erfüllt ist oder nicht. Die Bedingungen innerhalb einer Regel werden untereinander durch ein logisches UND verknüpft.

Der Maßnahmenanzeiger enthält die auszuführenden Maßnahmen. Ein 'x' bedeutet, daß diese Maßnahme ergriffen wird, wenn die Bedingungen der zugehörigen Entscheidungsregel erfüllt sind. Der ELSE-Teil ist optional. Er legt fest, welche Maßnahmen zu ergreifen sind, wenn keine andere Regel der Entscheidungstabelle zutrifft.

Bild 5 bringt ein Beispiel für die Formulierung einer Entscheidungstabelle mit EPOS-R.

Die Entscheidungstabelle wurde dabei aus der in EPOS-R vorliegenden Beschreibung (entspricht der oberen Hälfte des Bildes) generiert.

Die ganzen Zahlen im Maßnahmenanzeiger der Entscheidungstabelle legen die Reihenfolge fest, in der die Maßnahmen auszuführen sind.

Im **Begriffslexikon** werden Begriffe fachtechnisch geklärt, d.h. der Begriffsname wird mit dem zugehörigen Definitionstext in das Lexikon eingetragen. Damit soll zum einen die Begriffswelt unter den Projektbeteiligten vereinheitlicht werden und die Kommunikation gefördert werden. Zum anderen ist das Begriffslexikon für die Einarbeitung neuer Mitarbeiter und für das Projektmanagement von Bedeutung.

// Auszug aus dem Eingabe-Protokoll:

```

0007      DECISION-PROCESS  Motor-Steuerung
0008
0009      CONDITIONLIST :    ^Spannung OK^ (ja, nein),
0010                        Drehzahl (zu-hoch, normal, niedrig).
0011
0012      OPERATIONLIST :    LOOKAT ^Motor    abschalten^,
0013                        ^Motor    bremsen^,
0014                        ^Motor    beschleu.^,
0015
0016      RULES      :      (nein,    -    ; 1, 2, -),
0017                        ( ja, zu-hoch ; -, X, -),
0018                        ( ja, niedrig ; -, -, Y),
0019                        (      ,      ; -, -, -),
0020
0021      COMMENT "Der Entscheidungsprozess beschreibt den Zusammenhang
0022              zwischen der Netzspannung und Drehzahl mit den
0023              Steuereingriffen des Motors"

```

// Auszug aus dem Gesamt-Dokument:

DT : Motor-Steuerung

		1	2	3	ELSE
IC 1	Spannung OK	nein	ja	ja	
IC 2	Drehzahl	-	zu-hoch	niedrig	
IO 1	Motor abschalten, P. 2-6	1	-	-	-
IO 2	Motor bremsen	2	X	-	-
IO 3	Motor beschleu.	-	-	X	-

COMMENT :

Der Entscheidungsprozess beschreibt den Zusammenhang zwischen der Netzspannung und Drehzahl mit den Steuereingriffen des Motors

Bild 5: Beispiel für die Formulierung einer Entscheidungstabelle

Der Inhalt einer **identifizierbaren Aufgabenkomponente** kann mit natürlich-sprachlichen Texten oder mit Entscheidungstabellen beschrieben werden.

In EPOS-R wird zwischen zwei Typen von identifizierbaren Aufgabenkomponenten unterschieden:

- Aufgabenkomponenten vom Typ Anforderung (Requirement):
Dies sind Bedingungen, die vom Auftraggeber festgelegt werden.
- Aufgabenkomponenten vom Typ Voraussetzung (Constraint):
Dies sind zwingend einzuhaltende Randbedingungen, die sich aus technischen oder organisatorischen Gründen ergeben.

Neben der inhaltlichen Beschreibung und dem Schlüsselwort Requirement oder Constraint umfaßt die Beschreibung einer identifizierbaren Aufgabenkomponente noch eine Nummer. Mit dieser Nummer werden die Voraussetzungen und Anforderungen durchnummeriert. Eine zweite optionale Nummer kann in Klammern angegeben werden. Mit dieser Nummer wird das Kapitel in dem der formale Einschub definiert wurde angesprochen.

Beispiele für derart beschriebene Aufgabenkomponenten sind:

- ANFORDERUNG 9(2) <Lenkorgan, Verteilstation>

"Die Umschaltung eines Lenkorgans darf nur vorgenommen werden, wenn sich kein Paket in der dazugehörigen Verteilstation befindet."

- VORAUSSETZUNG 1(2)

"Am Eingang und Ausgang jeder Verteilstation sind Lichtschranken angebracht, die den Durchgang eines Paketes melden."

So beschriebene Aufgabenkomponenten können im Pflichtenheft oder während des konzeptionellen Entwurfs weiter verfeinert, d.h. ersetzt werden. Während des konzeptionellen oder operationellen Entwurfs können Aufgabenkomponenten auch quittiert werden. Wird eine Aufgabenkomponente bereits im konzeptionellen Entwurf quittiert, so braucht sie im operationellen Entwurf nicht mehr berücksichtigt zu werden. Das bedeutet nur, daß die Sektion in der die Quittierung angegeben wird, die entsprechende Anforderung oder Voraussetzung syntaktisch erfüllt. Die inhaltliche Erfüllung der Anforderung oder Voraussetzung kann nicht automatisch überprüft werden.

Mit **EPOS-A** sollen Spezifikationsfehler in der Aufgabenstellung und im konzeptionellen Entwurf aufgedeckt werden. Dazu werden die mit EPOS-R formulierten Eingaben auf syntaktische Richtigkeit und auf Verträglichkeit mit den Einträgen in der EPOS-Datenbank überprüft. Entscheidungstabellen werden auf Redundanz (sind identische Regeln enthalten), Eindeutigkeit (sind die Bedingungsanzeigerkombinationen aller Entscheidungsregeln disjunkt) und Vollständigkeit (sind alle elementaren Bedingungsanzeigerkombinationen durch die Entscheidungsregeln erfaßt) überprüft.

Die Ersetzung oder Quittierung identifizierbarer Aufgabenkomponenten in der Aufgabenstellung und im konzeptionellen Entwurf ist ein weiterer Prüfpunkt. Dabei wird ermittelt, ob die Ersetzungen oder Quittierungen vollständig sind, bzw. wird angezeigt, bei welchen ersetzten oder quitierten Aufgabenkomponenten die Spezifikation fehlt.

Das im Bild 3 angegebene Gliederungsschema für die Aufgabenstellung und das analoge Schema für den konzeptionellen Entwurf werden nach den entsprechenden Prüfungen in der EPOS-Datenbank gespeichert.

Aus diesen Daten werden mit **EPOS-D** rechnerunterstützt verschiedene Dokumente erzeugt wie z.B.

- eine Gesamtdokumentation mit Inhaltsverzeichnis
- eine Liste der identifizierbaren Aufgabenkomponenten
- das alphabetisch geordnete Begriffslexikon

Die Dokumente können bei einer geänderten Aufgabenstellung mit Hilfe von EPOS-D automatisch aktualisiert werden.

2.2.3. Die Spezifikationssprache EPOS-S

Die Spezifikationssprache EPOS-S dient zur Beschreibung des operationellen Entwurfs. EPOS-S enthält als formale Sprachmittel **Entwurfsobjekte**, **Kontrollflußkonstrukte** und **Elementaroperationen**. Bild 6 bringt einen Überblick über die EPOS-Entwurfsobjekte und die ihnen zugeordneten Schlüsselwörter und graphischen Symbole.

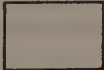
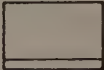
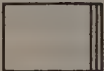
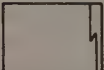
Entwurfsobjekt- Typ	Englisches Schlüsselwort	Deutsches Schlüsselwort	Graphische Darstellung (Symbole)
Aktion	ACTION	AKTION	
Modul	ACTION MODULE	AKTION MODUL	
Date	DATA	DATE	
Schnittstelle	INTERFACE	SCHNITTSTELLE	
Ereignis	EVENT	EREIGNIS	
Bedingung	CONDITION	BEDINGUNG	
Ausführungs- einheit	DEVICE	AUSFÜHRUNGS- EINHEIT	

Bild 6: EPOS-Entwurfsobjekte und die ihnen zugeordneten Symbole

Zu den graphischen Symbolen ist anzumerken, daß die Objekte bei der Dokumentation in einer graphischen Darstellung ausgehen werden können, jedoch gibt es keine graphische Eingabemöglichkeit.

Auf die Entwurfsobjekte wird noch genauer eingegangen, zunächst sollen jedoch die Begriffe Elementaroperationen und Kontrollflußkonstrukte geklärt werden.

Elementaroperationen sind Sprachmittel, die auf Entwurfsobjekte einwirken und den Kontrollfluß beeinflussen. Elementaroperationen können, wie der Name sagt, nicht weiter verfeinert werden.

nert werden. Aus der Liste der vordefinierten Elementaroperationen werden die datenabhängige Fortsetzung von Aktionen und der Abbruch von Aktionen. (Bild 7) angeführt.

Die Ai seien Aktionen, die Ci Bedingungen und die Di Daten.

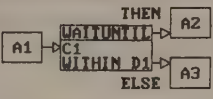
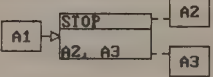
Operation	Beispiel	Bedeutung	Graphische Darstellung
Daten- abhangige Fortsetzung von Aktionen	A1; WAIT UNTIL C1 WITHIN D1 THEN A2 ELSE A3 WAITEND	Nach A1 warten bis Bedingung C1 erfullt ist, dann A2. Falls C1 innerhalb der Zeitdauer D1 nicht erfullt wird, dann A3.	
Abbruch von Aktionen	A1; STOP(A2, A3)	Nach Ausfuhrung von A1 werden die Verarbeitungs- vorgange A2 und A3 abgebrochen.	

Bild 7: Beispiele von Elementaroperationen in EPOS-S

Kontrollflukonstrukte sind Strukturelemente des Kontrollflusses. Als Beispiele fur Kontrollflukonstrukte zeigt Bild 8 die nebenlufige Aktion und die bedingte Verzweigung.

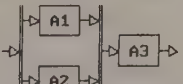
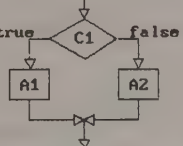
Kontrollflu- konstrukt	Beispiel	Bedeutung	Graphische Darstellung
Nebenlufige Aktionen	PARALLEL(A1, A2); A3	A1 und A2 laufen parallel ab. Wenn beide abgeschlossen sind folgt A3.	
Bedingte Verzweigung	IF C1 THEN A2 ELSE A3 FI	Falls Bedingung C1 erfullt ist, dann A1, sonst A2.	

Bild 8: Beispiele von Kontrollflukonstrukten in EPOS-S

Durch Datenfluß und Kontrollfluß kann die Verbindung zwischen den Entwurfsobjekten hergestellt werden. Als Datenfluß bezeichnet man die Übertragung von Daten zwischen Entwurfsobjekten. Der Kontrollfluß gibt die Reihenfolge an, in der Entwurfsobjekte angestoßen bzw. Elementaroperationen ausgeführt werden.

Jedes der **Entwurfsobjekte** aus Bild 6 wird nach folgendem allgemeinen Schema beschrieben:

Schlüsselwort, das den Typ des Entwurfsobjekts angibt	Name des Entwurfsobjekts
Beschreibungsteil	(Obligatorisch)
Verfeinerungsteil	(Optional)
Verbindungen zu anderen Entwurfsobjekten	(Optional)
Schlüsselwort, das das Ende der Beschreibung angibt.	

DESCRIPTION:	
PURPOSE:	Verbale Erläuterung der Aufgabe des Bausteins
NOTE:	Zusätzliche Entwurfsinformation
TEST:	Testplanung
PERFORMANCE:	Ausführungsanforderungen
DATE:	Terminangabe
FULFILS:	Liste der Anforderungen, die in EPOS-R festgelegt wurden und in diesem Baustein erfüllt werden.
CATEGORY:	Liste von Begriffen, die diesem Baustein zugeordnet sind
IDENTIFICATION:	Codewort zur anwendungsspezifischen Kennzeichnung dieses Entwurfsobjekts
DESCRIPTIONEND.	

Bild 9: Allgemeiner Aufbau der EPOS-S Beschreibung eines Entwurfsobjekts

Der Beschreibungsteil, beginnend mit dem Schlüsselwort DESCRIPTION, ist obligatorisch. In diesem Teil muß der mit dem

Schlüsselwort PURPOSE eingeleitete Abschnitt bearbeitet werden. Die übrigen Angaben sind optional.

Stellvertretend für alle Entwurfsobjekte soll das Entwurfsobjekt vom Typ Aktion genauer behandelt werden.

Mit dem Entwurfsobjekt vom Typ Aktion werden Vorgänge beschrieben, bei denen in einem bestimmten zeitlichen Ablauf Daten verknüpft, umgeformt oder verarbeitet werden.

ACTION	Bezeichner.
DESCRIPTION:	Beschreibungsteil
DESCRIPTIONEND:	
DECOMPOSITION:	Unteraktionen und Elementaroperationen, in die die Aktion verfeinert wird, einschliesslich der Angabe des Kontrollflusses zwischen den Unteraktionen bzw. der Angabe, ob die Unteraktionen unabhängig voneinander durch Ereignisse angestossen werden.
TRIGGERED:	Liste der Entwurfsobjekte vom Typ Ereignis, die die Aktion anstossen können.
INPUT:	Liste aller Eingangsdaten der Aktion (soweit sie auf dieser Entwurfsebene bekannt sind) ,
OUTPUT:	Ausgangsdaten der betrachteten Aktion.
REALIZATION:	Art der Realisierung der Aktion.
PROCESSED:	Ausführungseinheit, durch welche die Aktion ausgeführt wird.
ACTIONEND	

Bild 10: EPOS-S Beschreibung eines Entwurfsobjekts vom Typ Aktion

Bei den Schlüsselwörtern INPUT und OUTPUT kann nach der Angabe der Eingangs- bzw Ausgangsdaten die Schnittstelle zum technischen Prozeß beschrieben werden. Dazu muß nach dem Schlüsselwort FROM bzw. TO ein Bezeichner für ein Entwurfsobjekt vom Typ Schnittstelle stehen. Natürlich muß dieses Entwurfsobjekt vom Typ Schnittstelle und jedes weitere in der Aktionsbeschreibung verwendete Entwurfsobjekt ebenfalls durch eine EPOS-S Beschreibung spezifiziert werden.

Im nachfolgenden Beispiel einer EPOS-S Aktionsbeschreibung mit dem Namen 'Heizungsregelung-und-Überwachung' bedeuten die Symbole (/ und /), daß die innerhalb dieser Symbole aufgeführten Unteraktionen parallel ablaufen sollen.

ACTION HEIZUNGSREGELUNG-UND-UEBERWACHUNG

DESCRIPTION :

PURPOSE : "REGELUNG UND UEBERWACHUNG EINER HEIZUNGSANLAGE UNTER VERWENDUNG EINES MIKRORECHNERS ZU DEN AUTOMATISIERUNGSAUFGABEN GEGENÜBER DIE REGELUNG DER TEMPERATUREN IN 2 UNABHÄNGIGEN RAUMZONEN DURCH VERSTELLEN DER VORLAUFTEMPERATUREN, SOWIE DIE KESSELTEMPERATUR-REGELUNG UND DIE BRENNERUEBERWACHUNG NACH DER ANSTOSS DURCH EIN EXTERNES EREIGNIS 'SYSTEMSTART' SOLLEN ZUNÄCHST DIE SOLLWERTE EINGEGEBEN WERDEN (INITIALISIERUNG) DANN ERST SOLLEN DIE DREI REGELKREISE FREIGEgeben WERDEN, WOBEI DREI VERSCHIEDENE ABTASTZYKLEN (ZYKLUS-1, ZYKLUS-2 UND ZYKLUS-KESSEL) VERWENDET WERDEN

NOTE : "IN EINER SPÄTEREN AUSBAUSTUFE SOLLEN DIE SOLLWERTE TAGESZEIT ABHÄNGIG VORGEgeben UND GESPEICHERT WERDEN

TEST : "TESTPARAMETER SIEHE UNTERLAGE 25/100

DATE : 6 80

FULFILLS : REQUIREMENT 4 - 0

DESCRIPTIONEND

DECOMPOSITION : INITIALISIERUNG ,
SET (ZYKLUS-1 , ZYKLUS-2 , ZYKLUS-KESSEL , BRENNERSTÖRUNG , ANFORDERUNG)
(/ REGELUNG-1 , REGELUNG-2 , REGELUNG-KESSEL , SOLLWERT-EINGABE ,
BRENNER-UEBERWACHUNG /)

TRIGGERED : SYSTEMSTART

INPUT : REGELGRÖSSEN ,
SOLLWERTE FROM EINGABEGERÄT

OUTPUT : STELLGRÖSSEN TO SCHUETZE

REALIZATION : SOFTWARE IN 'PEARL'

PROCESSED : MIKRORECHNER

ACTIONEND

Bild 11: Beispiel für eine EPOS-S Aktionsbeschreibung

Die Beschreibung der übrigen Entwurfsobjekte erfolgt analog, nach dem in Bild 9 angegebenen allgemeinen Schema.

Ein Entwurfsobjekt vom Typ Modul wird eingeführt, wenn nicht der zeitliche Ablauf, wie bei der Aktion, sondern die logische Abgrenzung interessiert. Verschiedene Entwurfsobjekte werden in einem Modul nach bestimmten logischen Gesichtspunkten zusammengefaßt. Die Beschreibung eines Entwurfsobjekt vom Typ Modul enthält als wesentlichen Bestandteil die wirkungs-

mäßige Zuordnung zu anderen Moduln, d.h. die Angabe, welche Aktionen und Daten von anderen Moduln importiert bzw. an andere Moduln exportiert werden.

Sämtliche während des Entwurfsvorgangs auftretenden Daten, d.h. Informationen wie Stellgrößen, Meßwerte und Uhrzeiten, werden mit Entwurfsobjekten vom Typ **Date** beschrieben. Neben fest vorgegebenen Datentypen wie **BIT** für Bitketten oder **CLOCK** für Uhrzeiten können vom Anwender selbst Datentypen definiert werden.

In der Beschreibung eines Entwurfsobjekts vom Typ **Date** können Wertebereiche und Anfangswerte für variable Größen und für konstante, von Aktionen nicht veränderbare Größen zugewiesen werden. Bei variablen Größen kann zusätzlich eine Liste der Aktionen angegeben werden, von denen diese Daten verändert bzw. gelesen werden dürfen.

Der Datenaustausch zwischen dem zu entwerfenden technischen System und seiner Umwelt, also seinem 'übergeordneten System' (d.h. dem technischen Prozeß, dem Bedienpersonal), wird mit dem Entwurfsobjekt vom Typ **Schnittstelle** beschrieben. Bei der Beschreibung kann der Name der Ausführungseinheit angegeben werden durch die die Schnittstelle realisiert wird (z.B. Analogeingabewerk).

Ereignisse werden mit dem Entwurfsobjekt vom Typ **Ereignis** beschrieben. Ereignisse werden in Form von Binärsignalen gemeldet und beeinflussen den Ablauf von Aktionen. Es wird zwischen Unterbrechungen (Interrupts), zyklischen Ereignissen und Ereignissen zu bestimmten absoluten Zeitpunkten unterschieden.

Das Entwurfsobjekt vom Typ **Bedingung** dient zur Beschreibung datenabhängiger Bedingungen, die den Kontrollfluß von Aktionen steuern. Es kann die logischen Werte 'wahr' oder 'falsch' annehmen und durch einen logischen Ausdruck weiter verfeinert werden.

Mit Entwurfsobjekten vom Typ **Ausführungseinheit** werden Geräte oder Hardware-Einheiten, d.h. Einheiten, die Aktionen oder Schnittstellen ausführen, beschrieben.

Um Entwurfsfehler beim operationellen Entwurf aufzuzeigen, können wiederum mit Hilfe von **EPOS-A** bestimmte Prüfungen durchgeführt werden.

Die EPOS-S Beschreibungen werden dabei folgenden Prüfungen unterworfen:

- Eingaben werden auf syntaktische Richtigkeit überprüft
- die EPOS-S Beschreibungen werden auf Vollständigkeit überprüft; Namenskonflikte und Beschreibungsteile ohne Bezug zum Restsystem werden durch EPOS-A angezeigt
- entwurfsspezifische "Fehler" wie fehlerhafte Prozeduraufrufe, fehlende Wertzuweisung für Daten, Nicht-Einhaltung des Gültigkeitsbereichs von Daten, undefinierbare Angaben werden untersucht
- Überprüfung hierarchischer Eigenschaften: Typ-Konsistenz, hierarchische Vollständigkeit bei der Spezifikation der Ein- und Ausgabedaten von Aktionen, hierarchische Konsistenz des Gültigkeitsbereichs bei Daten
- es wird geprüft, ob optionale Beschreibungsteile vorhanden sind (vgl. 2.2.6 Verfahren der Codeselektion)
- Überprüfung der Synchronisierung, und schließlich
- wird geprüft, ob die in der Aufgabenstellung und im konzeptionellen Entwurf spezifizierten Anforderungen und Voraussetzungen, d.h. die identifizierbaren Aufgabenkomponenten, sofern sie nicht schon in EPOS-R quittiert wurden, in den EPOS-S Entwurfsobjekten quittiert, d.h. berücksichtigt wurden.

Der operationelle Entwurf kann rechnerunterstützt mit EPOS-D dokumentiert werden. Aus den eingegebenen EPOS-S Spezifikationen können automatisch z.B. Datenstrukturdiagramme nach Art von Michael Jackson, Flußdiagramme, Nassi-Shneiderman-Diagramme, Petrinetze (zur Darstellung paralleler, aber abhängiger Vorgänge) oder Hardware-Blockschaltbilder (zur graphischen Darstellung von Hardware-Einheiten) erzeugt werden.

Der Übergang von der EPOS-S Spezifikation zum Programmcode wird in Kapitel 2.2.6 ausführlich behandelt.

2.2.4. Die Spezifikationsprache EPOS-P und das Management Programmsystem EPOS-M

Die Spezifikationsprache EPOS-P enthält auf das Projektmanagement bezogene Sprachmittel, sogenannte Managementobjekte, mit denen Informationen für das Projektmanagement und die Produktverwaltung beschrieben werden können.

Für die rechnerunterstützte **Projektplanung** sind die Managementobjekte vom Typ Aktivität und Projektbeteiligter zu verwenden. Durch diese beiden Managementobjektarten kann die gesamte Projektstruktur (d.h. die Gesamtheit der wesentlichen Beziehungen zwischen den Elementen eines Projekts), die Netzplanung und die Projektorganisation beschrieben werden. Als Beispiel für ein Managementobjekt zeigt Bild 12 den allgemeinen Aufbau des EPOS-P Objekts vom Typ Projektbeteiligter. Mit diesem Objekt können die Funktionen, Zuständigkeiten und Kompetenzen der am Projekt beteiligten Personen, Abteilungen oder Firmen angegeben werden.

TEAMMEMBER	Name des Projektbeteiligten
Attributsangaben	Liste von Kennzeichnungs- bzw. Zusammenfassungsattributen, z.B. von Gruppen, Abteilungen
Funktion	Verbale Beschreibung der Funktion des Projektbeteiligten im Projekt
Zuständigkeiten	Liste der Aktivitäten, für die der Projektbeteiligte verantwortlich ist, bzw. die bearbeitet werden sollen
Kompetenzen	Angabe der Zugriffsrechte auf die EPOS-Datenbank
Strukturierung	Beschreibung der Stellung in der Projektorganisation
TEAMMEMBEREND	

Bild 12: Allgemeiner Aufbau der EPOS-P-Beschreibung des Managementobjekts vom Typ Projektbeteiligter

Mit dem Managementobjekt vom Typ Bericht sollen die dynamischen Daten, die während eines Projektablaufs entstehen, z.B. die angefallenen Kosten, erfaßt werden; damit können Entwicklung und Stand eines Projekts dargestellt werden.

Die Managementobjekte Fehlerbericht und Änderungsantrag sind formale Sprachmittel für die **Produktverwaltung**. Sie werden benötigt, um die verschiedenen Versionen des Entwurfs oder von Dokumenten etc. festzulegen und daran durchgeführte Änderungen zu kontrollieren.

Mit dem Managementprogrammsystem **EPOS-M** soll die **Projektführung** und die **Produktverwaltung** am Rechner unterstützt werden.

Mit EPOS-R und EPOS-P wurden Informationen in die EPOS-Datenbank eingegeben; dies waren z.B. Aufgabenstellung und Anforderung im Pflichtenheft, Definition von Teilprojekten, Definition von Arbeitspaketen im Projektstrukturplan, Projektbeteiligte, Aufstellung eines Organisationsplanes, Planung des zeitlichen Ablaufes sowie des zeitlichen Abflusses von Kapital, etc. Das Programmsystem EPOS-M wertet diese Daten aus und erzeugt daraus für die **Projektplanung** Projektstrukturpläne, Histogramme, Netzpläne und Matrizen zur Darstellung von Zuständigkeiten und Kompetenzen.

EPOS-M Berechnungsprogramme sammeln die geplanten Kosten und Ressourcen pro Zeiteinheit und berechnen die Terminplanung, die Zeitreserven und den kritischen Pfad in der Netzplanung. Die Projektplanungsdaten werden auf syntaktische Korrektheit, Konsistenz, Eindeutigkeit und Vollständigkeit überprüft.

Um den **Projektfortschritt** darzustellen, können mit EPOS-M geplante, in Arbeit befindliche und abgeschlossene Aufgaben mit Balkenplänen, Histogrammen oder Tabellen dokumentiert werden. Der Projektfortschritt kann außerdem mit Listen, Tabellen oder Diagrammen, die an vorher definierten Kontrollpunkten ausgegeben werden, beurteilt werden.

Die **Produktverwaltung** umfaßt bei einem Entwicklungsprojekt die Versions- und Variantenverwaltung. Versionen sind Weiterentwicklungen, als Varianten bezeichnet man verschiedene

alternative Ausprägungen von Teilen des Software-/Hardware-Systems.

Jedes EPOS-P Managementobjekt, jede EPOS-R Sektion und jedes EPOS-S Entwurfsobjekt enthält einen Versionsidentifikations-
teil, bestehend aus der Versionsnummer, dem Konfigurationsbe-
zeichner, dem Datum, dem Autor der Änderung oder der Eingabe
und schließlich Informationen über die Änderung.

Bestehende Versionen oder Konfigurationen können nach ver-
schiedenen Such- oder Auswahlkriterien übersichtlich in Li-
sten oder Tabellen dargestellt werden.

2.2.5. Die Bedienung des EPOS-Arbeitsplatzrechners mit EPOS-C

Die EPOS-R, EPOS-S und EPOS-P Beschreibungen können an einem
Sichtgerät eingegeben werden, die mit EPOS-D, EPOS-A und
EPOS-M erstellten Dokumentationen und Prüfergebnisse etc. kön-
nen ebenfalls am Bildschirm oder über Plotter und Drucker
ausgegeben werden.

Bei der Eingabe ist auch Stapelbetrieb möglich.

Der Benutzer kann bei der Eingabe unter drei verschiedenen
Bedienformen wählen. Diese geben mehr oder weniger ausführ-
lich die Bedienmöglichkeiten an, z.B. die Eingabealternativen
mit Menüs. Zudem kann der Benutzer mit Makros Bedienwünsche
zusammenfassen.

Das Arbeiten mit EPOS wird durch einen sogenannten Formular-
prozessor erleichtert. Bei der Eingabe neuer Spezifikationen
werden notwendige Schlüsselwörter automatisch eingefügt,
optionale werden angeboten. Bei Änderungen von EPOS-R, EPOS-S
und EPOS-P Spezifikationen können notwendige Schlüsselwörter
nicht gelöscht werden, neue Schlüsselwörter können nur an
syntaktisch richtigen Stellen eingefügt werden.

2.2.6 Der Übergang von der EPOS-S Spezifikation zum Pro- grammcode

Das EPOS-System stellt dem Anwender zwei Arten der automati-
schen Codeerzeugung zur Verfügung.

Bei dem Verfahren der Code Selektion mit dem Codeconnector

wird vom **Anwender** bei der Beschreibung eines Entwurfsobjekts im Code-Teil der Programmcode in der Zielsprache selbst formuliert. Die auf die verschiedenen Entwurfsobjekte verteilten Informationen werden dann zu einem korrekten Quellprogramm zusammengesetzt. Der Code-Teil steht bei der Beschreibung von Aktionen, die nicht weiter verfeinert werden, anstelle des DECOMPOSITION-Parts. (siehe Bild 10)

Bei dem Verfahren der automatischen Codegenerierung werden die EPOS-Kontrollflußkonstrukte, Daten-, Ereignis- und Bedingungsspezifikationen automatisch in Ausdrücke einer höheren Programmiersprache umgesetzt, soweit dies überhaupt sinnvoll möglich ist. Die Umsetzung kann nur für Programmiersprachen erfolgen, deren Syntax und Semantik dem Codegenerator bekannt sind. Derzeit liegen "Nullversionen" für die Codegenerierung in PASCAL und für den Problemtail von PEARL vor; die Programmgeneratoren sind jedoch noch nicht an das EPOS-System angekoppelt. /EPOS85b/

Bei EPOS-Konstrukten, die nicht in entsprechende Programmkonstrukte umzusetzen sind, wird vom Codegenerator eine Fehlermeldung ausgegeben. Dies betrifft z.B. Echtzeit-Konstrukte bei der Umsetzung nach PASCAL.

Um den Codegenerator leicht an neue Programmiersprachen anpassen zu können, wurden die Vorschriften für die automatische Umsetzung formal beschrieben.

Der Codegenerator erzeugt nur bedingt effiziente Programme, da die kontextunabhängigen EPOS-Konstrukte nach einer Standardmethode, d.h. nach fest vorgegebenen Regeln, umgesetzt werden.

Im erzeugten Programmcode kann jedoch vom Anwender optimiert werden, der optimierte Code wird mit dem Entwurf verglichen und Änderungen werden im Entwurf automatisch nachgezogen. Damit wird die Konsistenz zwischen Spezifikation und Programm Quellcode sichergestellt.

Beide Verfahren, d.h. Codeselektion und Codeumsetzung, können getrennt benutzt werden. Bei der integrierten Methode wird immer dann Codeselektion durchgeführt, wenn ein Entwurfsobjekt einen Code-Teil enthält, die Codeumsetzung wird gewählt, wenn das Entwurfsobjekt einen Verfeinerungsteil besitzt.

Für die automatische Codeumsetzung in ein ablauffähiges Programm wird eine **vollständige** EPOS-S Spezifikation gefordert.

Die meisten EPOS-S Sprachkonstrukte können eindeutig in PEARL-Anweisungen umgesetzt werden, da die EPOS-S Sprachkonstrukte an die Syntax und Semantik von PEARL angelehnt sind.

Im folgenden werden einige Regeln für die automatische Codegenerierung für PEARL beschrieben. Es wird aufgezeigt, wie die Aktionsstruktur einer EPOS-S Spezifikation in ein PEARL-Programm abgebildet wird. Die Umsetzung der EPOS-Kontrollflußkonstrukte und Elementaroperationen auf PEARL-Konstrukte wird an einigen Beispielen behandelt. /EPOS85b/

Das Entwurfsobjekt vom Typ **Aktion** wird in einen **PEARL-Block**, d.h. einen Modul, eine Task oder eine Prozedur, umgesetzt. Die Umsetzung von EPOS-S Teilen in BEGIN-END Blöcke der Programmiersprache PEARL wird in der momentanen Version nicht berücksichtigt.

Die Aufteilung der Tasks und Prozeduren auf **PEARL-Module** muß vom Benutzer im interaktiven Betrieb selbst vorgenommen werden. Als Richtlinie für die Aufteilung wird u.a. vorgeschlagen, funktionell und datenmäßig miteinander verknüpfte Tasks und Prozeduren in einen PEARL-Modul zusammenzufassen.

PEARL-Tasks können in EPOS-S u.a. am TASK-Attribut erkannt werden. Außerdem müssen die im PARALLEL-Part aufgeführten Aktionen (siehe Bild 8 Kontrollflußkonstrukt: Nebenläufige Aktion) in Tasks umgesetzt werden. Ebenso sind Aktionen, die in der auf den STOP-Part folgenden Liste stehen, (siehe Bild 7 Elementare Operation: Abbruch von Aktionen) in PEARL Tasks abzubilden.

Aktionen für die im Verfeinerungsteil das Schlüsselwort **USING** und eine zugehörige Datenliste angegeben sind, werden in **Prozeduren** umgesetzt. Nach dem Schlüsselwort **USING** werden die aktuellen Parameter der Prozedur aufgeführt.

Beispiel:

ACTION UVW.

.

.

DECOMPOSITION: A;

XYZ USING (D11,D12)

.

.

In einer späteren Version sollen Aktionen, die im Verfeinerungsteil von mindestens zwei weiteren Aktionen stehen als Prozedur realisiert werden.

In der Aktion, die eine Prozedur darstellt, werden in EPOS die formalen Eingabe- und Ausgabe-Parameter (nach den Schlüsselwörtern INPUT und OUTPUT) aufgeführt und die Zuordnung zwischen formalen und aktuellen Parametern (nach dem Schlüsselwort ALT) beschrieben.

Beispiel:

ACTION XYZ.

.

.

INPUT: FP1,

FP2.

OUTPUT: FP2.

ALT: FP1 = D11,

FP2 = D12.

.

.

ALT: FP1 = DN1,

FP2 = DN2.

ACTIONEND

Diese Aktion wird in folgende PEARL-Prozedur umgesetzt. (Die Typangaben bei den formalen Parametern müssen noch der zugehörigen Datenspezifikation entnommen werden.)

```
XYZ: PROCEDURE (FP1<typ>,FP2<typ>IDENT);
```

```
/* Vereinbarungen und  
Anweisungen */
```

```
END;
```

Die im INPUT- bzw OUTPUT-Teil der als Prozedur erkannten Aktion aufgeführten Parameter werden per Adresse übergeben, in PEARL durch das Schlüsselwort IDENT gekennzeichnet.

Beim Aufruf der Prozedur werden die nach dem Schlüsselwort USING aufgeführten Daten als aktuelle Parameter an die PEARL-Prozedur übergeben. Der Aufruf der Prozedur erfolgt an der Stelle, an der die Prozedur im Verfeinerungsteil eines EPOS-Entwurfsobjekts steht.

Beispiel:

```
/* BEGIN ACTION UVW */  
/* BEGIN ACTION A */  
/* Verfeinerungen von A */  
/* END ACTION A */  
  
CALL XYZ (D11,D12);  
  
/* END ACTION UVW */
```

Die Abbildung der EPOS-Kontrollflußkonstrukte und Elementaroperationen auf PEARL-Konstrukte wird am Beispiel der nebenläufigen Aktion und der bedingten Verzweigung beschrieben.

Die Umsetzung der **nebenläufigen Aktion** soll am Beispiel eines PARALLEL-Parts innerhalb einer Sequenz aufgezeigt werden. Dazu müssen für die Synchronisation zusätzlich Semaphorvariablen (SEMAB2 .. SEMABM) und Semaphoroperationen (REQUEST, RELEASE) generiert werden.

EPOS :

PEARL :

ACTION XYZ.

DECOMPOSITION: A;
PARALLEL (B1,
B2,...,BN);
C.

ACTIONEND

/* END ACTION A */
ACTIVATE B2;
ACTIVATE BN;
/* BEGIN ACTION B1 */

/* END ACTION B1 */
REQUEST SEMAB2;

REQUEST SEMABN;
/* BEGIN ACTION C */

BM: TASK; /* M = 2,...,N */
/* ACTION BM */
RELEASE SEMABM;

END;

Die **bedingte Verzweigung** in EPOS-S wird in eine IF-THEN-ELSE-Anweisung in PEARL umgesetzt.

In EPOS-S kann die Verzweigungsbedingung durch ein Entwurfsobjekt vom Typ Bedingung (CONDITION) spezifiziert werden. Der logische Ausdruck ergibt sich aus dem Verfeinerungsteil dieses Objekts. In PEARL muß der Verzweigungsausdruck dagegen eine BIT(1)-Größe sein. Bei der Umsetzung wird daher der Verfeinerungsteil des Entwurfsobjekts vom Typ Bedingung direkt in die PEARL-Anweisung eingesetzt.

EPOS :

PEARL :

ACTION IJK.

DECOMPOSITION:

IF WEITER THEN A
ELSE B

FI;

ACTIONEND

CONDITION WEITER.

DECOMPOSITION: STATUS NE 'VOLL'.

CONDITIONEND

IF STATUS NE 'VOLL'
THEN A
ELSE B
FIN;

Die Elementare Operation **Abbruch von Aktionen** in EPOS wird durch eine TERMINATE-Anweisung in PEARL realisiert. Die in der Liste nach dem STOP-Part angegebenen Aktionen müssen deshalb PEARL-Tasks entsprechen.

Als weiteres Beispiel für die Umsetzung von Elementaroperationen wird die datenabhängige Fortsetzung von Aktionen, auch als WAIT-UNTIL-Konstrukt bezeichnet, betrachtet.

Der Ablauf des Kontrollflusses ist beim WAIT-UNTIL-Konstrukt abhängig von der Bedingung C1. Dieser logische Wert muß bei der Umsetzung nach PEARL zyklisch in einer Schleife abgefragt werden.

EPOS:

DECOMPOSITION:

```
A1;  
WAIT UNTIL C1  
    WITHIN D1  
THEN A2  
    ELSE A3  
WAITEND;  
A4;
```

PEARL:

ABC : TASK;

```
/* END ACTION A1 */  
BEGIN  
    DCL WAITTIME DUR INITIAL (...);  
    DCL WAITZAEHLER DUR INITIAL (0 EINHEIT);  
    WHILE (NOT C1) AND (WAITZAEHLER LT WAITTIME ) REPEAT  
        AFTER ... EINHEIT RESUME;  
        WAITZAEHLER := WAITZAEHLER + EINHEIT;  
    END;  
END;  
IF C1 THEN /* BEGIN ACTION A2 */  
    /* END ACTION A2 */  
    ELSE /* BEGIN ACTION A3 */  
        /* END ACTION A3 */  
FIN;  
/* BEGIN ACTION A4 */
```

Die RESUME-Anweisung im Schleifenrumpf stellt sicher, daß der Wert der logischen Bedingung C1 durch andere Tasks verändert werden kann. (Durch RESUME wird die laufende Task unterbrochen. Nach Ablauf der Zeitdauer, die hinter dem Schlüsselwort

AFTER angegeben ist, wird die Task wieder in den Zustand laufend versetzt.)

Die Schleife wird durchlaufen, solange die Bedingung C1 nicht 'wahr' ist, und die Wartezeit D1 nicht überschritten ist.

In der IF-THEN-ELSE-Anweisung, die auf die Schleife folgt, wird in Abhängigkeit vom Wert der Bedingung C1 eine Anweisungsfolge durchlaufen, die dem THEN oder ELSE Teil im WAIT-UNTIL-Konstrukt entspricht.

Mit diesen Beispielen soll das Prinzip bei der Codeumsetzung erklärt werden. Es soll auch gezeigt werden, daß dem Programmierer durch die Codeumsetzung wesentliche Teile der Synchronisation abgenommen werden.

In /EPOS85a/ sind Grenzen, Mängel und Schwachstellen zusammengestellt, die durch den industriellen Einsatz von EPOS gesammelt wurden:

- Bei automatisch gezeichneten Diagrammen wird viel Platz verschenkt und die Schriftgröße ist viel zu klein.
- Im Verfeinerungsteil von Entwurfsobjekten sollten auch Boole'sche Ausdrücke zugelassen sein.
- Die Textdokumentation von EPOS-R, EPOS-S und EPOS-P sollte gleich sein.
- Die Robustheit des EPOS-Systems läßt zu wünschen übrig.
- Die Bedienung ist zu "holprig" und zu unbequem.
- Zur Erleichterung der Anwendung fehlt eine graphische Eingabemöglichkeit.
- Man bräuchte eine Schnittstelle, um andere Werkzeuge mit EPOS koppeln zu können.
- Zusätzlicher Aufwand für die schritthaltende Dokumentation, vor allem in den ersten Phasen eines Projekts.

2.3. Software-Entwicklung mit Hilfe eines Transformations-Expertensystems

Software-Entwicklung wird in Phasen unterteilt. Als Ergebnis dieser Phasen entstehen Dokumente. Ganz allgemein läßt sich sagen, daß die Dokumente späterer Phasen durch Transformation aus Dokumenten früherer Phasen entwickelt werden können. In diesem Kapitel soll eine transformationelle Software-Entwicklungsumgebung, bestehend aus Benutzerschnittstelle, Werkzeugen und Datenbasis, vorgestellt werden. /PERS85/

Zu den Werkzeugen zählt als wesentliche Komponente neben konventionellen Werkzeugen wie Editor oder Compiler das **Transformationssystem**. Die Regeln, die zur Transformation angewendet werden, sind in der Datenbasis abgelegt.

Die Datenbasis enthält darüber hinaus die Dokumente, die während der Software-Entwicklung entstehen. Die Dokumente werden jedoch nicht in textueller Form, sondern strukturiert gehalten. An die Werkzeuge wird damit die Anforderung gestellt, auf Strukturen und nicht mehr auf Texten zu arbeiten. Der Benutzer kann mit einem Editor Dokumente erstellen oder verfeinern. Mit **Analysatoren** werden die Dokumente auf Konsistenz und Vollständigkeit untersucht, mit dem **Transformationssystem** können sie von einer Phase im Software-Lebenszyklus zur nächsten transformiert werden.

Die Transformation kann interaktiv erfolgen oder automatisch ablaufen. Im interaktiven Fall werden die Regeln explizit vom Benutzer angegeben.

Im folgenden wird das Transformationssystem TREX beschrieben. Anschließend wird die Anwendung der transformationellen Entwicklungsumgebung für Prozeßsteuerungssoftware gezeigt.

Soweit bekannt ist TREX noch nicht implementiert, daraus lassen sich wohl einige Unklarheiten in der nachfolgenden Darstellung erklären.

2.3.1. Das Transformationssystem TREX

Das Transformationssystem TREX ist nach den Prinzipien von Expertensystemen aufgebaut und besteht aus den Komponenten **Regelbasis**, **Regelauswerter** und **Erklärungskomponente**.

Die **Regeln** in der Regelbasis werden in folgender Form angegeben:

```
rule regelname : eingabe => ausgabe cond bedingung.
```

Die Regeln haben die Bedeutung, daß eine Struktur, die auf die eingabe (das Eingabe-Muster) paßt, zur ausgabe (zum Ausgabe-Muster) transformiert wird, wenn die Bedingung erfüllt ist.

Mit dem Regelnamen können die Regeln strukturiert werden. Dies entspricht einem Modulkonzept für Regeln und soll die Anwendung und den Entwurf von Regeln erleichtern.

Der Regelname kann Parameter enthalten; damit ist es möglich, Metaregeln zu formulieren, die die Anwendung von Regelmengen steuern. (siehe spätere Beispiele)

Bedingungen bestehen aus Konjunktion, Disjunktion und Negation von Prädikaten.

Als Beispiel wird eine Regelbasis für die Vereinfachung von Ausdrücken angegeben:

(Die Regeln werden numeriert, um bei den Erklärungen leichter auf die einzelnen Regeln eingehen zu können.)

```
1.) rule simple : plus (X,0) => X.
2.) rule simple : times (X,1) => X.
3.) rule simple : OP # [C1,C2] => C3
    cond constant (C1) and constant (C2) and
    eval (OP,C1,C2) = C3
```

Die Regel 3.) beschreibt, daß ein zweistelliger Ausdruck mit den konstanten Operanden C1 und C2 durch das Ergebnis C3 ersetzt wird. Dazu wurde ein Muster für zweistellige Operatoren benutzt. Allgemein haben Muster für "Funktoren" die Form

```
'variable # argumente'.
```

Variable beginnen mit einem Großbuchstaben.

Eine zweite Regelbasis zu obigem Beispiel der Vereinfachung von Ausdrücken beschreibt die Strategie der Regelanwendung von innen nach außen:

```

4.) rule inner(R) : F # [X] => R: F # [inner(R): X] .

5.) rule inner(R) : F # [X,Y] => R: F # [inner(R): X,
                                     inner(R): Y].

6.) rule inner(R) : X => X cond atom (X).

7.) rule simple_exp : Exp => inner(simple) : Exp.

```

Die inner-Regeln haben als Parameter die Regelmenge R. Diese ist von innen nach außen anzuwenden, d.h. bei der Regel 4 wird zuerst inner(R) auf das Argument X der Struktur F angewendet. Auf diesen Term wird die Regelmenge R angewendet, d.h. das Ausgabemuster von Regel 4 ist eine Regelanwendung. Muster können nämlich Regelanwendungen enthalten. Das bedeutet aber, daß an dieser Stelle das Ergebnis der Transformation eingesetzt wird.

Folgendes Beispiel soll die Regelanwendung verdeutlichen:
 (Als Regelname wird simple eingesetzt, der zu vereinfachende Ausdruck sei times(a,1). Die Nummer der Regel, die bei der Transformation angewendet wird, wird am rechten Rand in die Symbole /* */ eingeschlossen angegeben. Das Ausgabemuster rechts vom Symbol => wird gleichzeitig als Eingabemuster für die nächste Transformation verwendet.)

```

simple_exp : times (a,1)                               /* Regel 7 */
=> inner(simple) : times (a,1)                         /* Regel 5 */

=> simple : times # {inner(simple):a,                  /* Regel 6 */
                    inner(simple):1}                  /* Anwendung von
                                                    innen nach
=> simple : times # [a,1]                               außen      */

= simple : times (a,1)                               /* Gleichsetzung */

=> a                                                    /* Regel 2 */

```

Allgemein muß der **Regelauswerter** eine Menge von Regeln auf Daten anwenden. Dabei müssen Muster mit Daten verglichen werden und die Bedingungen ausgewertet werden, falls der Mustervergleich erfolgreich war.

Des weiteren muß die Reihenfolge, in der die Regeln auf die Daten angewendet werden sollen, definiert werden. Wie eben gezeigt, wird die Regelanwendung in TREX durch innere Regelanwendung oder durch Metaregeln, die der Benutzer angibt, gesteuert.

Das TREX-System verwendet als Strategie bei der Regelauswertung Tiefensuche mit Rücksetzen, d.h. bei einem Mißerfolg bei der Suche wird automatisch zur letzten Regelanwendung zurückgesetzt.

Bei einem Mißerfolg, also beim Scheitern einer Transformation, sollte der Benutzer unterrichtet werden, so daß er interaktiv eingreifen kann. Diese Aufgabe wird in TREX von der **Erklärungskomponente** übernommen. Diese Komponente hat sich auch die erfolgreich durchgeführten Transformationen zu merken, so daß diese wiederholt oder überarbeitet werden können.

2.3.2. Anwendung von TREX für Prozeßsteuerungssoftware

TREX soll für Prozeßsteuerungssoftware einsetzbar sein. Das zugrundegelegte Phasenmodell umfaßt die Anforderungsanalyse, die Spezifikation, Prototyping und Implementierung.

Für die Anforderungsdefinition wird die Sprache LARS eingesetzt; sie besteht aus den drei Teilen Guard (zur zentralen Steuerung der Aktivitäten), den Stimulus-Response-Netzen (SRM-Netze, die aufgrund eines Ereignisses einen "Response" erzeugen) und den "Interfaces" (den Schnittstellen zum technischen Prozeß). Daten und die zugehörigen Operationen werden während der Anforderungsdefinition in natürlicher Sprache beschrieben, während der Spezifikation durch algebraische Spezifikation formalisiert.

In der Prototyping-Phase wird die Spezifikation mit Hilfe von Transformationsregeln in ein ADA-Programm transformiert.

Der ADA-Prototyp enthält Komponenten, die die Teile der Spezifikation widerspiegeln.

3. Automatisches Programmieren

Bereits in der Einleitung dieser Arbeit wurden Begriffe wie automatische Umsetzung und Programmsynthese verwendet. Im letzten Kapitel wurde im Zusammenhang mit TREX von Transformation und automatischer Transformation gesprochen.

In diesem Kapitel sollen diese Begriffe und Methoden unter den Begriff "Automatisches Programmieren" eingeordnet werden. Zunächst werden verschiedene Definitionen für Automatisches Programmieren vorgestellt und geklärt, was in dieser Arbeit unter Automatischem Programmieren zu verstehen ist. Anschließend werden die grundsätzlichen Methoden zur Computer-Unterstützung des Programmiervorgangs aufgezeigt und ein Beispiel für Programmtransformation und Programmsynthese angegeben.

3.1. Begriffsklärung und Motivation

Aus den verschiedenen Definitionen für Automatisches Programmieren (AP) wurden willkürlich einige ausgewählt: /BARR82/
Eine Definition umschreibt Automatisches Programmieren recht einfach als irgendetwas, das den Programmierer von Routinearbeit oder unangenehmer Arbeit beim Programmieren entlastet.

Eine andere Definition besagt, daß ein AP-System bei einer gegebenen Beschreibung des zu lösenden Problems Teile der Programmiertätigkeit übernimmt; gemeint sind dabei Aktivitäten, die üblicherweise bei der Entwicklung eines Programms von einem Programmierer ausgeführt werden. Das Wesentliche derartiger AP-Systeme liegt also darin, daß sie vom menschlichen Problemlöser Teilaufgaben abnehmen und damit den Arbeitsbereich des Programmierers einschränken.

Eine dritte Definition sagt aus, AP bedeute, den Computer zum Schreiben seiner eigenen Programme einzusetzen.

Zusammenfassend könnte man Automatisches Programmieren als die Automatisierung von Teilen des Programmiervorgangs bezeichnen.

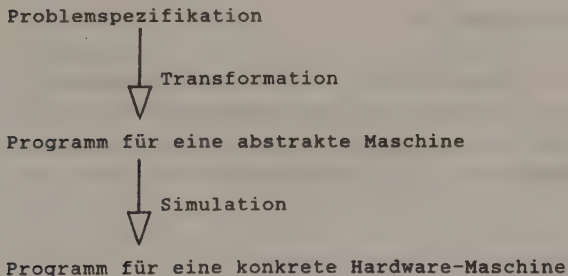
In dieser Arbeit sollen unter Automatischem Programmieren Methoden und Werkzeuge für computerunterstütztes Programmie-

ren verstanden werden.

Mit Automatischem Programmieren soll dem Programmierer die Problemlösung in einer höheren und natürlicheren Ebene ermöglicht werden, d.h. Problemlösung auf Spezifikationsebene und nicht auf Programmebene. Automatisches Programmieren als Teil einer Programmierumgebung soll den Programmierer am Computer unterstützen, damit schneller und leichter programmiert werden kann und vor allem bezüglich der Spezifikation korrekte Programme erstellt werden.

3.2. Möglichkeiten zur Computerunterstützung der Programmierung

Um die Möglichkeiten der Computerunterstützung bei der Programmierung aufzuzeigen, wird nach /RETT84/ der Weg von der Spezifikation zum ausführbaren Programm in zwei Teilschritte zerlegt. Ausgehend von der Problemspezifikation kommt man durch Transformation zunächst zu einem Programm für eine abstrakte Maschine. Die Ausführung dieses Programms wird auf der konkreten Hardware-Maschine 'simuliert'. Diesem Übergang entspricht Kompilation bzw. Interpretation. Die folgende Skizze soll diese Beschreibung verdeutlichen:



Es gibt zwei Arten, mit denen der Programmiervorgang automatisiert werden kann. Das ist zum einen die Entwicklung immer höherer Programmiersprachen mit den entsprechenden Compilern oder Interpretern. In der obigen Skizze bedeutet das, daß der Übergang von der Spezifikation zum Programm für die abstrakte Maschine kürzer wird. Für den Problemlöser bedeutet das, daß

dieser Übergang leichter durchzuführen ist. Dafür werden allerdings die Werkzeuge für den Übergang zur konkreten Maschine komplexer und komplizierter. Auf diese Möglichkeit zur Computerunterstützung des Programmiervorgangs wird im folgenden nicht eingegangen. Im weiteren wird die zweite Möglichkeit zur Automatisierung des Programmiervorgangs, nämlich die Computer-Unterstützung der Transformation, behandelt. Für die Computer-Unterstützung des Transformationsvorgangs, d.h. der Transformation der Problemspezifikation in ein Programm für eine abstrakte Maschine, werden im wesentlichen drei Paradigmen angegeben: /RETT84/

1.) Automatische Programmsynthese

Gegeben ist dabei ein **Grundwissen** und eine **Problemspezifikation**; gesucht wird ein Programm, das die in der Problemspezifikation beschriebenen Eigenschaften hat.

2.) Automatische Programmtransformation

Gegeben ist ein **Grundwissen** und ein **Programm'**; gesucht ist ein **Programm''**, das mit dem **Programm'** äquivalent ist. Zudem soll das **Programm''** (bzgl. irgendeines Kriteriums) besser als das **Programm'** sein.

3.) Automatische Verifikation

Gegeben ist ein **Grundwissen**, eine **Problemspezifikation** und ein **Programm für eine abstrakte Maschine**. Mit automatischer Verifikation soll geklärt werden, ob das Programm die in der Problemspezifikation angegebenen Eigenschaften hat.

3.3. Programmtransformation und Programmsynthese

Da in den weiteren Kapiteln auf Verifikation nicht näher eingegangen wird, wird im folgenden nur Programmtransformation und Programmsynthese anhand von Beispielen erklärt.

3.3.1. Programmtransformation

Programmtransformation wird häufig verwendet, um einfach geschriebene, leicht verständliche Programme in effizientere, dafür aber möglicherweise schlechter lesbare Programme umzusetzen.

Als Beispiel für Programmtransformation wird ein Teil eines Systems beschrieben, mit dem rekursive Programme in iterative umgeformt werden können. /BARR82/

Die Grundidee liegt darin, daß Teile eines Programms mit einem Eingabemuster, das ein Rekursionsschema enthält, verglichen werden. Ist dieser Mustervergleich erfolgreich und sind zudem bestimmte Vorbedingungen erfüllt, wird das rekursive Programm durch ein iteratives ersetzt. Dazu wird folgende Transformationsregel benötigt:

Eingabemuster: $f(x): \text{if } a \text{ then } b \text{ else } h(d, f(e));$

Vorbedingung: h ist assoziativ, x kommt nicht frei in h vor;

Ausgabemuster: $f(x):$

```
if a
then result <- b
else begin
  result <- d;
  x <- e;
  while not a
  do begin
    result <- h(result, d);
    x <- e
  end;
  result <- h(result, b)
end
```

Als Beispiel wird eine Funktion zur Berechnung der Fakultät angegeben:

```
FACTORIAL(x): if (x=1) then 1 else TIMES(x,FACTORIAL(x-1))
```

```
FACTORIAL(x): if (x=1)
               then result <- 1
               else begin
                   result <- x;
                   x <- (x-1);
                   while not (x=1)
                       do begin
                           result <- TIMES(result,x);
                           x <- (x-1)
                       end;
                   result <- TIMES(result,1)
               end
```

Programmtransformation und Programmsynthese sind eng miteinander verbunden, da Transformationsregeln auch in Synthesystemen eingesetzt werden. Mit Synthesystemen wird systematisch ein Programm aus einer gegebenen Spezifikation hergeleitet.

3.3.2. Programmsynthese

Im folgenden soll ein Synthesystem beschrieben werden, das die Transformationsregeln direkt auf die Problemspezifikation anwendet. /MANN79/

Grundidee dieses Systems ist, automatische Existenzbeweise bei der automatischen Programmsynthese anzuwenden, um damit nicht nur die Korrektheit von Programmen nachzuweisen, sondern gleichzeitig, praktisch als Nebenprodukt, ein Programm zu konstruieren.

Das System erfordert eine Spezifikationsmethode, die später mit einem geeigneten Regelsatz ausgewertet werden kann. Akzeptiert werden Spezifikationen mit sehr mächtigen Konstrukten, mit denen ohne Angabe eines Algorithmus ein Pro-

gramm spezifiziert werden kann.

Als Beispiel soll die Spezifikation zur Berechnung des größten gemeinsamen Teilers zweier nichtnegativer ganzer Zahlen x und y angegeben werden:

```
gcd(x y)  <=  compute max { z: z|x and z|y }  
           where x and y are nonnegative integers and  
                 x ≠ 0 or y ≠ 0
```

Das Synthesystem versucht so vorliegende Spezifikationen mit Transformationsregeln, ohne weiteres menschliches Eingreifen, in ein bezüglich der Spezifikation korrektes Programm zu transformieren.

Das erzeugte Programm ist korrekt und terminiert, Effizienz ist an dieser Stelle nicht von Bedeutung (siehe dazu Programmtransformation). Im Prinzip können Programme in einer beliebigen Zielsprache erzeugt werden, im nachfolgenden Beispiel wird eine LISP-ähnliche Sprache verwendet.

Bevor die Synthese am Beispiel beschrieben wird, wird zunächst auf Form, Art und Bedeutung der Transformationsregeln eingegangen.

Durch Transformationsregeln wird ein Teil einer Programmbeschreibung durch eine äquivalente andere Beschreibung ersetzt.

Es gibt drei Arten von Transformationsregeln:

Einige Regeln beinhalten "Wissen" über das zugrundeliegende Anwendungsgebiet (zu obigem Beispiel $\text{gcd}(x y)$ wären dies Eigenschaften von Integerzahlen). Eine zweite Gruppe behandelt die Abbildung der Konstrukte aus der Spezifikation in die Zielsprache. Die dritte Gruppe wird zur Formulierung von grundlegenden Programmkonstrukten (wie bedingte Anweisungen, Rekursion, etc.), und zwar unabhängig vom Anwendungsgebiet, eingesetzt.

Transformationsregeln werden in der Form

$$t \Rightarrow t'$$

angegeben, mit der Bedeutung, daß ein Ausdruck t durch den entsprechenden Ausdruck t' ersetzt werden kann. Die umgekehr-

te Richtung darf nur dann angewendet werden, wenn explizit eine Regel der Form $t' \Rightarrow t$ existiert.

Eine Regel der Form

$$t \Rightarrow t' \text{ if } P$$

bedeutet, daß die Transformation t nach t' nur dann durchgeführt werden darf, wenn die Bedingung P wahr ist.

Die Transformationsregel der Form

$$\text{true and } Q \Rightarrow Q$$

bedeutet, daß ein Ausdruck der Form $\text{true and } Q$ immer durch Q ersetzt werden kann.

Die in einer bestimmten Situation anwendbaren Regeln werden durch Mustervergleich ausgewählt. Können mehrere Regeln in einer bestimmten Situation angewendet werden, so muß das Synthesystem für die Auswahl einer anzuwendenden Regel sorgen.

Aus der Menge der anwendbaren Regeln kann z.B. durch eine vorgegebene Ordnung in den Regeln eine bevorzugt ausgewählt werden. Eine andere Möglichkeit liegt darin, daß jede Regel sogenannte strategischen Bedingungen enthält, die das sinnlose Anwenden einer Transformationsregel verhindern. Schließlich enthält das Synthesystem noch einen Backtracking-Mechanismus, der nach Anwendung einer Regel, die **keinen** Programmteil in der Zielsprache liefert, Rücksetzen und Fortfahren mit einer anderen anwendbaren Regel ermöglicht. (gemäß der vordefinierten Ordnung in den Regeln)

Durch Anwendung einer Transformationsregel auf eine gegebene Programmbeschreibung erhält man eine neue Programmbeschreibung, die auch als Ziel bezeichnet wird. Durch wiederholtes Anwenden der Transformationsregeln kommt man also von einem Ziel zum anderen und schließlich zum gewünschten Programm.

Das erste Ziel erhält man aus der Programm-Spezifikation. In obigem Beispiel $\text{gcd}(x\ y)$ wäre dies

$\text{compute max } \{ z : z|x \text{ and } z|y \}$

Im folgenden soll am Beispiel des $\text{gcd}(x\ y)$ gezeigt werden, wie mit diesem Synthesystem das Terminieren eines rekursiven Programms bewiesen wird, wobei es gleichzeitig kon-

struiert wird.

Da für diese Arbeit mehr die Konstruktion des Programms und weniger der Beweis zur Terminierung von Bedeutung ist, wird auf letzteres weniger eingegangen.

Es werden folgende Transformationsregeln benötigt:

Regel 1: $u|v \Rightarrow \text{true} \quad \text{if } v = \emptyset$

Regel 2: $u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|w - v$

Regel 3: $\max \{ u: u|v \} \Rightarrow v \quad \text{if } v \text{ is a positive integer}$

Regel 4: $P \text{ and } Q \Rightarrow Q \text{ and } P$

Regel 5: **condition formation rule**

Erklärung:

Ist ein Ziel der Form **prove P** weder zu beweisen noch zu widerlegen, so kann mit Hilfe dieser Regel eine Fallunterscheidung durchgeführt werden; d.h. die Fälle, in denen P wahr oder P falsch ist, können getrennt betrachtet werden.

Ein erfolgreich konstruiertes Programmsegment s1, das unter der Annahme "P sei wahr" das gegebene Problem löst, und ein entsprechendes Programmsegment s2 für "P sei falsch" können zur bedingten Anweisung

if P then s1 else s2

zusammengesetzt werden.

Regel 6: **recursion formation rule**

Erklärung:

Angenommen, es wäre ein Programm zu folgender Spezifikation

$f(x) \Leftarrow \text{compute } P(x) \quad \text{where } Q(x)$

zu entwickeln.

(Q(x) sei eine Bedingung, P(x) irgend ein Ausdruck in der Spezifikation)

Gibt es zur Spezifikation $\text{compute } P(x)$ ein Unterziel $\text{compute } P(t)$, dann kann man versuchen dieses Unterziel durch den rekursiven Aufruf $f(t)$ zu erreichen.

($f(x)$ soll ja $P(x)$ für jedes beliebige x , das $Q(x)$ erfüllt, berechnen)

Für die berechnungsfähige Einführung des rekursiven Aufrufs müssen allerdings zwei Bedingungen bewiesen werden:

- 1.) Die Eingabebedingung $Q(t)$, die sicherstellt, daß das Argument t des rekursiven Aufrufs $f(t)$ die (für das Zielprogramm) geforderte Eingabebedingung erfüllt
- 2.) die Terminierungsbedingung

Wie gezeigt, kann das erste Ziel für die Programmsynthese direkt aus der Spezifikation übernommen werden.

Ziel 1: $\text{compute max } \{ z: z|x \text{ and } z|y \}$

In dieser Situation sind zwei Regeln auf den Teilausdruck $z|x$ and $z|y$ anwendbar; dies sind die "logische" Regel 4

($P \text{ and } Q \Rightarrow Q \text{ and } P$) und die "numerische" Regel 2

($u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|w - v$).

Mit Hilfe der (hier nicht näher erläuterten) strategischen Bedingungen und der in den Regeln vorgegebenen Ordnung wird vom Synthesystem die Regel 4 ausgewählt. Damit erhält man als Unterziel

Ziel 2: $\text{compute max } \{ z: z|y \text{ and } z|x \}$

Ziel 2 entspricht dem Ziel 1, x und y sind lediglich vertauscht.

In dieser Situation kann daher wie oben beschrieben die Regel 6 (recursion formation rule) angewendet werden; d.h. es wird versucht, mit dem rekursiven Aufruf $\text{gcd}(y \ x)$ das Ziel 2 zu erfüllen. Die Anwendung dieser Regel fordert aber, daß dazu die Eingabebedingungen (Ziel 3) und die Terminierungsbedingung (Ziel 4) zu beweisen sind.

Ziel 3: prove y and x are nonnegative integers and
 $y \neq 0$ or $x \neq 0$

Ziel 4: prove $\text{gcd}(y\ x)$ terminates

Ziel 3 erhält man aus der Eingabebedingung der Spezifikation, es ist lediglich x and y durch y and x zu ersetzen.

Um das Terminieren eines rekursiven Programms $f(x)$ mit den rekursiven Aufrufen $f(t_1)$, $f(t_2)$, ... $f(t_n)$ zu beweisen, muß gezeigt werden, daß x , t_1 , t_2 , ... t_n zu einer wohlgeordneten Menge mit der Relation $\prec$ gehören, und daß

$t_1 \prec x$, $t_2 \prec x$, ... $t_n \prec x$ ist

Die nichtnegativen Integerzahlen bilden mit der Ordnung $\prec$ eine solche wohlgeordnete Menge.

Auf das genaue Verfahren und den Beweis wird an dieser Stelle nicht näher eingegangen. Es sollte durch diese kurze Beschreibung lediglich deutlich gemacht werden, daß es zum Beweis der Terminierung (Ziel 4) ausreicht

Ziel 5: prove $y \prec x$

zu beweisen. ($f(x) = \text{gcd}(x\ y)$, $f(t_i) = \text{gcd}(y\ x)$)

Für Paare von Argumenten, d.h. nichtnegativen ganzen Zahlen, ist folgende Ordnungsrelation definiert:

$(x_1\ x_2) \prec (y_1\ y_2)$ if $x_1 < y_1$,
 or if $x_1 = y_1$ and $x_2 < y_2$.

Ziel 5 kann aber nicht bewiesen werden, da x und y Eingabevariablen sind, deren Größe nicht bekannt ist. Nach obigem Regelsatz kann in einem derartigen Fall die **condition formation rule** mit Fallunterscheidung angewendet werden.

Fall $y \prec x$: In diesem Fall wird sowohl die Eingabebedingung (Ziel 3) als auch die Terminierungsbedingung (Ziel 4) erfüllt.

Mit den bisher bewiesenen Zielen kann das folgende Programmstück in der Zielsprache formuliert werden:

```

gcd (x y)  <=  if y < x
                then gcd(y x)
                else ...

```

Zur Konstruktion des else-Zweiges muß zu obiger Fallunter-
scheidung die Situation $x \leq y$ betrachtet werden:

Fall $x \leq y$: In diesem Fall darf der rekursive Aufruf $\text{gcd}(y\ x)$ nicht erfolgen, da die Terminierungsbedingung nicht erfüllt ist.

Mit den Zielen 3 bis 5 sollte das Ziel 2 bewiesen werden. Dies ist für den Fall $y < x$ gelungen, für den Fall $x \leq y$ nicht gelungen. Da es für Ziel 2 keine weitere Regel gibt, deren Anwendung zu einem Programmsegment in der Zielsprache führt; so muß zum Ziel 1 zurückgegangen werden und versucht werden, dort eine alternative Regel anzuwenden.

Bei Ziel 1 kann die alternative numerische Regel 2 angewendet werden. Eine nochmalige Anwendung der Regel 4 wäre sinnlos, da sonst Sequenzen folgender Form konstruiert würden:

P and Q, Q and P, P and Q, ...

Strategische Bedingungen schließen allgemein die wiederholte Anwendung von Regeln auf Teilausdrücke, die von den Regeln selbst produziert wurden, aus.

Durch Anwendung der numerischen Regel 2 auf Ziel 1 erhält man das

Ziel 6: `compute max { z: z|x and z|y - x }`

Ziel 6 ist dem Ziel 1 wiederum sehr ähnlich, x und y sind lediglich durch x und $y - x$ ersetzt. An dieser Stelle kann wieder die recursion formation rule mit dem rekursiven Aufruf

`gcd (x    y - x)`

angewendet werden um Ziel 6 zu erreichen. Dazu sind die Eingabebedingung (Ziel 7) und die Terminierungsbedingung (Ziel 8) zu beweisen.

Ziel 7: `prove x and y - x are nonnegative integers and`
 `x ≠ 0 or y - x ≠ 0`

Ziel 8: `prove gcd(x    y - x) terminates`

Der Beweis von Ziel 7, d.h. daß x und $y - x$ nichtnegative Integerzahlen sind, folgt zum einen aus der Spezifikation und zum anderen aus dem gerade zu betrachtenden Fall $x \leq y$. Es bleibt noch zu beweisen, daß $x \neq 0$ oder $y - x \neq 0$ ist. Dies kann durch Ziel 9 oder Ziel 10 bewiesen werden.

Ziel 9: prove $x \neq 0$

Ziel 10: prove $y - x \neq 0$

Ziel 9 kann ebenso wie Ziel 10 weder bewiesen noch widerlegt werden, d.h. es kann wiederum die condition formation rule angewendet werden:

Fall $x \neq 0$: In diesem Fall ist die Eingabebedingung für den rekursiven Aufruf $\text{gcd}(x \ y - x)$ erfüllt.

Zum Beweis der Terminierung wird das Wissen, daß $x > 0$ ist, $y-x$ eine nichtnegative Interzahl ist und daß damit $y-x < y$ ist verwendet. Jede Ausführung eines rekursiven Aufrufs verkleinert das zweite Argument, so daß nur eine endliche Zahl von Ausführungen möglich ist, bis das zweite Argument 0 ist. Damit ist die Eingabebedingung (Ziel 7) und die Terminierungsbedingung (Ziel 8) bewiesen und es kann ein Teilprogramm folgender Form konstruiert werden:

```
gcd (x y)  <= if y < x
              then gcd (y x)
              else if x ≠ 0
                  then gcd (x y - x)
                  else ...
```

Zur Konstruktion des else-Zweiges muß zu obiger Fallunterscheidung noch der Fall $x = 0$ betrachtet werden.

Fall $x = 0$: In diesem Fall darf die recursion formation rule nicht angewendet werden, da wiederum die Terminierungsbedingung nicht erfüllt werden kann. Das bedeutet aber, daß auf Ziel 6 zurückgegriffen werden muß und dort eine alternative Regel angewendet werden muß.

Aus Ziel 6 kann für den Fall $x = 0$

Ziel 11: **compute max { z: z|0 and z|y }**

formuliert werden.

Nach Anwendung der Regel 1 kann Ziel 11 durch

compute max { z: true and z|y }

ersetzt werden. Dies entspricht

compute max { z: z|y }

Darauf ist die Regel 3 anwendbar, aus dieser erhält man

Ziel 12: **compute y.**

Damit konnte durch automatische Programmsynthese ein vollständiges Programm für den gcd (x y) in der Zielsprache formuliert werden:

```
gcd (x y) <=  if y < x
               then gcd (y x)
               else if x ≠ 0
                     then gcd (x y-x)
                     else y.
```

Bezüglich anderer Anwendungsgebiete erscheint es aber fraglich, ob sich derartig geeignete und präzise Transformationsregeln finden lassen.

4. Die Spezifikationsmethode PASS

In diesem Kapitel wird die Spezifikationsmethode PASS (Parallel Activity Specification Scheme), soweit diese zum Verständnis der nachfolgenden Kapitel benötigt wird, vorgestellt.

PASS wurde von A. Fleischmann im Rahmen einer Dissertation an der Universität Erlangen entwickelt. /FLEI84/

PASS ist eine Spezifikationsmethode, die zur Spezifikation von verteilten Systemen, d.h. Automatisierungssystemen, DFÜ-Protokollen etc. geeignet ist. PASS besteht aus zwei Teilen: Dies sind die Beschreibung der Kommunikationsstruktur und die Definition des Prozeßsystems.

Im Kapitel 4.1 wird der Aufbau der Kommunikationsstruktur beschrieben. Im Kapitel 4.2 wird angegeben, wie jede einzelne Strukturierungseinheit der Kommunikationsstruktur weiter zu verfeinern ist. /KRAG84/

Kapitel 4.3 bringt als Zusammenfassung das Gliederungsschema für die Spezifikation. /HEIL85/

4.1. Beschreibung der Kommunikationsstruktur

In der Kommunikationsstruktur wird beschrieben, welche Hardware- und Software-Prozesse an einem Prozeßsystem beteiligt sind und welche Beziehungen zwischen diesen Prozessen bestehen, d.h. es wird angegeben, welche Nachrichten zwischen Prozessen ausgetauscht werden.

Graphisch werden die beteiligten Prozesse, im folgenden auch Strukturierungseinheiten genannt, durch Rechtecke dargestellt. Ein Pfeil, beschriftet mit dem Namen der Nachricht(en), zeigt vom sendenden zum empfangenden Prozeß.

Bild 13 enthält die Strukturierungseinheiten A, B und C; A sendet die Nachricht X an die Strukturierungseinheit C und die Nachricht Y an die Strukturierungseinheit B, B schließlich sendet die Nachricht Z an C.

Jede Strukturierungseinheit in der graphischen Darstellung kann ein einzelner Prozeß, genannt **Einzelprozeß** (im Bild 13 sind dies die Strukturierungseinheiten A und B), eine **Prozeßgruppe** oder ein **Prozeßbündel** (Strukturierungseinheit C)

sein. Prozeßgruppen und Prozeßbündel können selbst wieder eine Menge von einzelnen Prozessen enthalten. Die Prozesse eines Prozeßbündels oder einer Prozeßgruppe dürfen zusätzlich zur Kommunikation über Nachrichten noch über gemeinsame Objekte kommunizieren. Die Prozesse eines Prozeßbündels dürfen von außen, d.h. von einer anderen Strukturierungseinheit angesprochen werden. Für die Prozesse einer Prozeßgruppe ist dieses Ansprechen von außen verboten.

Es kann die Kommunikationsstruktur zwischen Typen von Strukturierungseinheiten und die Kommunikationsstruktur zwischen konkreten Strukturierungseinheiten dargestellt werden. Jeder Typ einer Strukturierungseinheit, d.h. genauer eines Softwareprozesses, muß in einem zweiten Schritt (in der Definition des Prozeßsystems) genauer beschrieben werden.

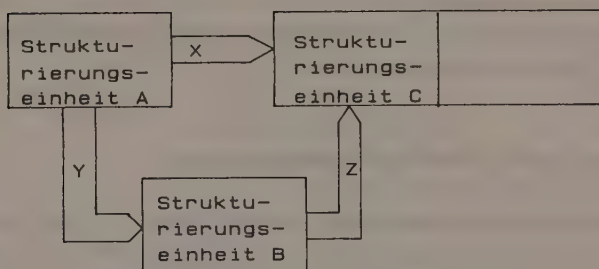


Bild 13: Beispiel einer Kommunikationsstruktur

4.2. Beschreibung der Definition des Prozeßsystems

Die Struktur eines Einzelprozesses umfaßt die Kommunikationsmaschine, die Ablaufsteuerung und die private Benutzermaschine. Jeder einzelne Prozeß aus einer Prozeßgruppe oder aus einem Prozeßbündel wird ebenfalls nach diesem Schema beschrieben. Zusätzlich muß bei Prozeßbündeln und Prozeßgruppen für die Kommunikation über gemeinsame Objekte noch die ge-

meinsame Benutzermaschine definiert werden.

Im Anschluß an die Beschreibung der Typen können für jede Strukturierungseinheit beliebig viele Objekte dieses Typs deklariert werden.

4.2.1. Die Kommunikationsmaschine

Die Kommunikationsmaschine regelt das Senden und Empfangen von Nachrichten. Die Eigenschaften der Kommunikationsmaschine sind fest vorgegeben. Bei der Spezifikation kann hier lediglich angegeben werden, ob die Nachrichten synchron oder asynchron ausgetauscht werden sollen; d.h. es kann spezifiziert werden, ob die Nachrichten über einen sogenannten Wartebereich (Puffer) ausgetauscht werden sollen und wie groß dieser sein soll.

4.2.2. Die Ablaufsteuerung

In der Ablaufsteuerung wird für einen Prozeß festgelegt, in welcher Reihenfolge Nachrichten mit welchem Partnerprozeß ausgetauscht werden und wann interne Berechnungen durchgeführt werden.

Die Ablaufsteuerung wird graphisch mit den Elementen Knoten und Kanten dargestellt.

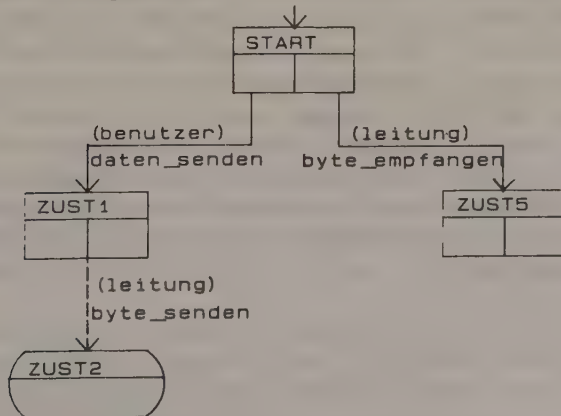


Bild 14: Ausschnitt aus einer Ablaufsteuerung

In einem Kommunikationsknoten (Rechtecksymbol) können Nachrichten gesendet oder empfangen werden. Die Sendekanten (gestrichelte Kanten) sind jeweils mit dem Namen des Adressaten (im Bild 14 z.B. der Prozeßname 'leitung') und dem Namen der zu sendenden Nachricht (z.B. byte_senden) beschriftet. Die Empfangskanten (durchgezogene Kanten) sind mit dem Absender und dem Namen der erwarteten Nachricht versehen. Kann die Nachricht gesendet werden, bzw. trifft die erwartete Nachricht ein, so wird in den durch die Kante gekennzeichneten Folgezustand übergegangen. Führen von einem Kommunikationsknoten mehrere Sende- oder Empfangskanten weg, so wird alternativ auf Nachrichten gewartet (d.h. es wird auf irgendeine der angegebenen Nachrichten gewartet), bzw. werden alternativ Nachrichten gesendet (Bild 14 Zustand mit Namen START). Ein gleichzeitiges Warten auf mehrere Nachrichten bzw. ein gleichzeitiges Senden von Nachrichten läßt sich durch eine entsprechende Kantenbeschriftung (das Zeichen & zwischen den einzelnen Nachrichten) darstellen.

In der graphischen Darstellung der Ablaufsteuerung werden interne Berechnungen durch Internknoten (Ovale) gekennzeichnet. In diesen Ovalen wird der Name der auszuführenden internen Berechnung eingetragen. Es wird zwischen internen Operationen, die die lokalen Daten eines Prozesses verändern, und internen Funktionen, die den Zustand der lokalen Daten abfragen, unterschieden. Einen Internknoten mit einer internen Operation verlassen Operationskanten (gestrichelte Kanten), diese sind mit den möglichen Resultaten der Operationsausführung beschriftet. Von einem Internknoten mit einer internen Funktion führen sogenannte Funktionskanten (durchgezogene Kanten) weg. Die Funktionskanten tragen die möglichen Ergebnisse der internen Funktion. Abhängig vom Ergebnis wird in einen durch die Kante bezeichneten Folgezustand übergegangen. Bisher wurde festgelegt, wann welche Aktionen ausgeführt werden. Die Definition dieser Aktionen erfolgt in der privaten Benutzermaschine.

4.2.3. Die private Benutzermaschine

Die private Benutzermaschine enthält die lokalen Daten eines Prozesses, die Definition der internen Funktionen und Operationen und die Beschreibung der Ausgabefunktionen und Eingabeoperationen.

Jeder Nachricht, die gesendet wird, wird eine Ausgabefunktion zugeordnet. Diese Funktion ermittelt aus dem internen Zustand eines Prozesses die Werte der Nachrichtenparameter.

Für jede Nachricht, die empfangen wird, muß eine Eingabeoperation definiert werden. Diese beschreibt, wie sich der Empfang einer Nachricht auf den weiteren Zustand eines Prozesses auswirkt.

Sämtliche Funktionen und Operationen können mit Techniken spezifiziert werden, wie sie in der sequentiellen Programmierung üblich sind, z.B. umgangssprachliche Beschreibung, pseudoprogrammiersprachliche Beschreibung oder algebraische Spezifikation.

4.2.4. Die gemeinsame Benutzermaschine

Prozeßbündel und Prozeßgruppen bestehen aus mehreren Prozessen, die über gemeinsame Objekte verfügen. Diese Objekte werden in der gemeinsamen Benutzermaschine definiert. Der Zugriff auf die Objekte kann in der gemeinsamen Benutzermaschine durch ressourcenorientierte Synchronisationskonzepte, z.B. Semaphore, geregelt werden.

4.3. Gliederungsschema der Spezifikation

PASS-Spezifikationen unterliegen einer feststehenden Gliederung. Das Gliederungsschema wird auf der folgenden Seite angegeben. Zunächst sind noch einige darin verwendete Begriffe zu klären:

In der Liste der Kommunikationspartner (Gliederung I.4) werden die Strukturierungseinheiten mit den Namen der jeweiligen Kommunikationspartner und den Namen der auszutauschenden Nachrichten aufgeführt.

Der Punkt I.3 betrifft die Kommunikation zwischen Prozessoren. In der graphischen Darstellung dieser Kommunikationsstruktur werden die Prozessoren durch Rechtecke dargestellt. Die Softwareprozesse werden dem Prozessor zugeordnet, auf dem sie später ausgeführt werden sollen. In der graphischen Darstellung werden die Software-Prozesse im zugehörigen Prozessor-Rechteck aufgelistet. Die Pfeile sind mit den Nachrichten, die zwischen den Prozessoren fließen, beschriftet.

Gliederungsschema der Spezifikation

- I Kommunikationsstruktur
 - I.1 Kommunikation zwischen Typen von Strukturierungseinheiten
 - I.2 Kommunikation zwischen konkreten Strukturierungseinheiten
 - I.3 Kommunikation zwischen Prozessoren
 - I.4 Liste der Kommunikationspartner

II Definition des Prozesssystems

	Prozessbündel	Prozessgruppen	Einzelprozesse
II.1	II.1.1	II.1.2	II.1.3
Typen	A. Prozesse (1,2,...) 1. Kommunikationsmaschine 2. Ablaufsteuerung 3. private Benutzermaschine B. gemeinsame Benutzermaschine	A. Prozesse (1,2,...) 1. Kommunikationsmaschine 2. Ablaufsteuerung 3. private Benutzermaschine B. gemeinsame Benutzermaschine	A. Prozesse (1,2,...) 1. Kommunikationsmaschine 2. Ablaufsteuerung 3. private Benutzermaschine B. gemeinsame Benutzermaschine
II.2	II.2.1	II.2.2	II.2.3
Dekl.	Prozesse (1,2,...)	Prozesse (1,2,...)	Prozesse (1,2,...)

5. Eine Programmierumgebung für verteiltes PEARL

In diesem Kapitel wird die Programmierumgebung für verteiltes PEARL (PU) in groben Zügen vorgestellt.

Nach der Einordnung der Programmierumgebung in den Software-Lebenszyklus wird zunächst die Sicht des Benutzers auf die Programmierumgebung gezeigt. Anschließend wird die interne Sicht beschrieben, d.h. ein Überblick über die Programmteile, Datenstrukturen und Schnittstellen gegeben. Eine genaue Beschreibung der einzelnen Bestandteile erfolgt in den Kapiteln 6 bis 8.

Gemäß dieser Beschreibung wird im Rahmen von drei Studienarbeiten /GRAD86/, /HERM / und /BAUR / ein Prototyp an einem IBM kompatiblen PC erstellt.

Die Zielsprache PEARL (Process and Experiment Automation Realtime Language) ist die Sprache der Prozeßrechner. PEARL enthält Sprachkonstrukte zur Echtzeitverarbeitung, zur Kommunikation mit einem Technischen Prozeß, zur Tasksteuerung und Synchronisation von Prozessen. Unter **verteilt**em PEARL ist im wesentlichen AVIONIC-PEARL mit den in Erlangen durchgeführten Erweiterungen, dies sind Sprachkonstrukte zur Kommunikation über Botschaften und nichtdeterministische Kontrollanweisungen, zu verstehen. /FLEI83/
Beispiele für diese Sprachkonstrukte werden in Kapitel 8 angegeben.

5.1. Einordnung in den Software-Lebenszyklus

Im Software-Lifecycle nach /KIMM79/ (siehe Bild 14) unterstützt die Programmierumgebung für verteiltes PEARL die umrandeten Phasen. Mit der Programmierumgebung wird der Benutzer sowohl in der Spezifikationsphase als auch beim Übergang zum dokumentierten Programm, durch Programmsynthese, unterstützt.

Die Programmierumgebung enthält weder Editor noch Compiler, Binder oder Lader. Sie stellt auch keine Werkzeuge für die Anforderungsdefinition, für Verifikation oder Test, Versionskontrolle und Management zur Verfügung. Der Hauptaspekt

liegt vielmehr auf der Erzeugung von PEARL-Code, deshalb wurde die Bezeichnung Programmierungsumgebung und nicht Software-Produktionsumgebung gewählt.

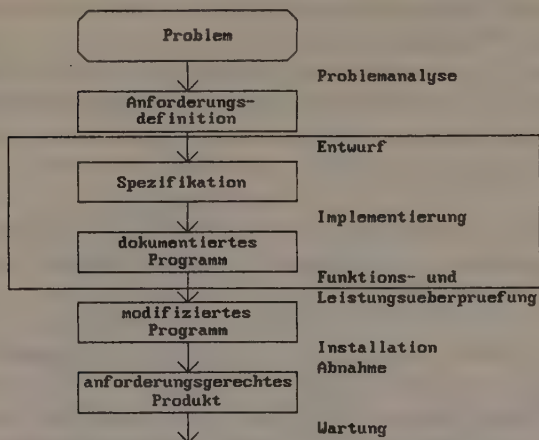


Bild 14: Der Software-Lebenszyklus

5.2. Benutzersicht

Aus der Sicht des Benutzers stellt die PU für die Spezifikation einen Graphikeditor (Zeichenprogramm) sowie Masken und Menüs für die nichtgraphischen Teile der Spezifikationsmethode zur Verfügung. Die Programmsynthese kann vom Benutzer im Anschluß an eine vollständige Spezifikation angestoßen werden.

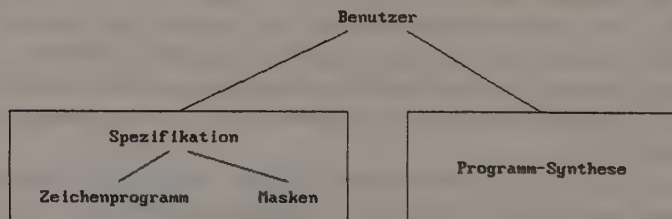


Bild 15: Die Programmierungsumgebung aus der Sicht des Benutzers

Der Benutzer kann mit dem **menüorientierten Zeichenprogramm** die graphischen Teile der Spezifikation in PASS, d.h. die Kommunikationsstruktur und die Ablaufsteuerung, am Bildschirm erstellen. Die Bilder können auch auf einem Plotter ausgegeben werden.

Sollen diese graphischen Teile der Spezifikation als Eingabe für die Programmsynthese verwendet werden, so hat sich der Benutzer bereits beim Erstellen der Bilder an bestimmte Richtlinien zu halten; d.h. der Benutzer darf nur **vordefinierte graphische Objekte** verwenden. Als Beispiel sollen hier vordefinierte Symbole für Kommunikationsknoten und Internknoten in der Ablaufsteuerung oder Symbole für Prozeßbündel, Prozeßgruppen oder Einzelprozesse in der Kommunikationsstruktur angeführt werden.

In dieser Arbeit wurden sämtliche die Spezifikationsmethode PASS betreffenden Bilder mit dem Zeichenprogramm und den eben angesprochenen vordefinierten Symbolen erstellt.

Eine Zusammenstellung aller vordefinierten Symbole findet sich im Anhang 2.

Für den Benutzer ist die Richtlinie, nur vordefinierte graphische Symbole in den Zeichnungen zu verwenden, keine Einschränkung. Sind ihm die Namen und die Bedeutung der Symbole bekannt, so können die Kommunikationsstruktur und die Ablaufsteuerung relativ einfach erstellt werden. Das Layout der Bilder bleibt nach wie vor in der Hand des Benutzers. Die folgenden Bilder sollen die Vorgehensweise beim Einfügen eines Kommunikationsknotens in ein bereits vorhandenes Teilbild verdeutlichen.

Im Bild 16 soll ein neuer Kommunikationsknoten eingefügt werden. Der Benutzer hat dazu die Stelle, an der das Symbol eingefügt werden soll, zu markieren und das Kommando zum Einfügen eines Symbols zu geben. Anschließend wird von ihm der Name des einzufügenden Symbols erfragt. In diesem Fall wird vom Benutzer der Name "kommkno" eingegeben. Bild 17 ist das Ergebnis dieser Aktionen.

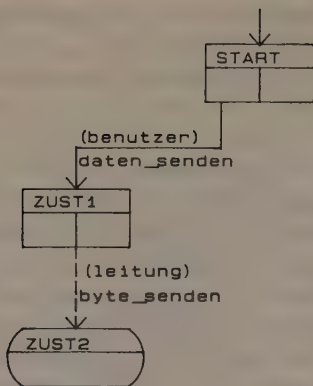


Bild 16: Ausschnitt aus einer Ablaufsteuerung

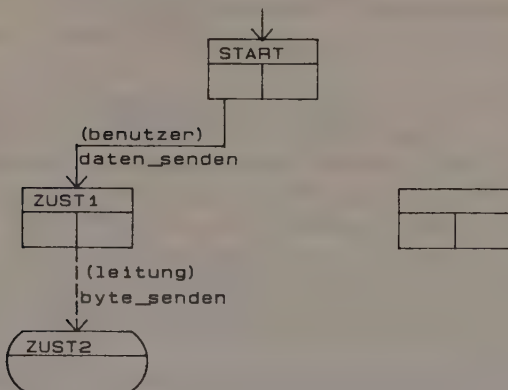


Bild 17: Ausschnitt aus einer Ablaufsteuerung

Ein Nachteil des verwendeten Zeichenprogramm /ZP86/ für die Programmierungsumgebung liegt darin, daß mit dem Zeichenprogramm grundsätzlich beliebige graphische Objekte erzeugt werden können. Beim Zeichnen eines Bildes kann zudem nicht überprüft werden, ob sich der Benutzer an oben erwähnte Vereinbarungen hält. Bei der Auswertung der Bilddaten kann dann lediglich festgestellt werden, daß das Bild nicht identifizierbare

Objekte enthält und für die weitere Verarbeitung in der PU unbrauchbar ist.

Nach dem Erstellen der PASS-Graphiken kann der Benutzer die nichtgraphischen Teile der Spezifikationsmethode bearbeiten. Der Benutzer wird dabei durch die Definition des Prozesses mit vordefinierten **Masken** und Menüs geführt. Die Masken werden in einem Fenster am Bildschirm angeboten, zuvor werden sie mit Informationen aus den graphischen Teilen der Spezifikation gefüllt. Die folgende Zeichnung soll den Bildschirmaufbau verdeutlichen:

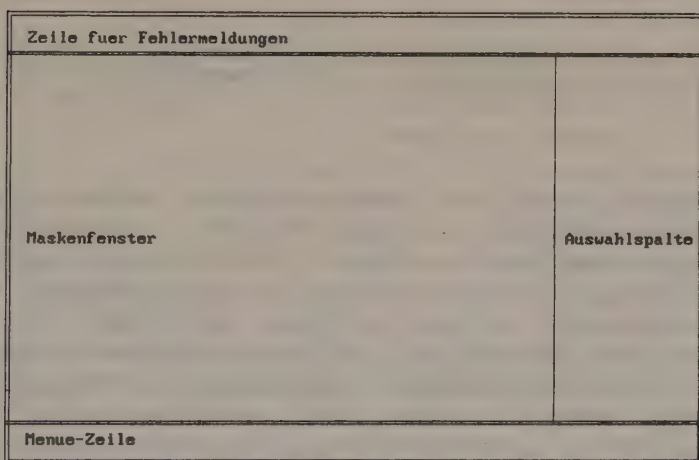


Bild 18: Aufbau einer Maske

In der Menüleiste werden sämtliche zu einer Maske möglichen Kommandos übersichtlich dargestellt.

Dem Benutzer wird in den Masken vorgegeben, an welchen Stellen Text eingegeben werden kann. Er kann zu jeder Zeit von ihm bereits eingefügte Texte ändern, löschen oder ergänzen, d.h. die Spezifikation kann schrittweise geändert und vervollständigt werden.

Die Arbeitsweise mit den Masken soll folgendes Beispiel verdeutlichen.

Zeile fuer Fehlermeldungen	
Maske 9: Private Benutzermaschine	
II.1.1 Typen von Einzelprozessen	
<u>proto</u>	
Ausgabefunktionen: <u>byte_senden</u>	
a. umgangssprachliche Beschreibung	
b. Pseudocode	
c. PEARL-Code	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Bild 19: Beispiel einer Maske

Diese Maske enthält ausreichend Information, um den Benutzer über das momentan zu bearbeitende Teilstück in der Spezifikation zu unterrichten. Im Bild 19 wird der Benutzer darüber informiert, daß er momentan Typen von Einzelprozessen und davon speziell den Prozeß mit Namen 'proto' zur weiteren Bearbeitung ausgewählt hat. In der gezeigten Maske soll die zum Prozeß 'proto' gehörende Ausgabefunktion 'byte_senden' genauer spezifiziert werden. Dazu kann der Benutzer in der Maske wählen, ob er die Teile a.), b.) oder c.) bearbeiten möchte. Er kann diese Punkte in beliebiger Reihenfolge bearbeiten und Text eintragen. Er kann die gesamte Sitzung beenden und zu einer Vorgänger- oder Nachfolgermaske blättern. Die möglichen Kommandos sind in der Menüzeile angegeben. Eine Zusammenstellung aller Masken mit den jeweils möglichen Kommandos findet sich im Anhang 3.

Im Anschluß an eine vollständige Spezifikation kann der Benutzer die Programmsynthese anstoßen. Eventuell notwendige Ergänzungen im erzeugten PEARL-Programmgerüst sollen wiederum mit Hilfe von Masken und Menüs vom Benutzer erfragt werden.

Grundsätzlich können mehrere Benutzer an der Erstellung einer

Spezifikation für ein bestimmtes Prozeßsystem arbeiten. Mehrere Benutzer können z.B. an verschiedenen Arbeitsplatzsystemen die graphischen Teile der Spezifikation erstellen. Die Auswertung der Bilder und die weitere Bearbeitung der Spezifikation kann jedoch nicht über mehrere Arbeitsplatzsysteme verteilt erfolgen.

5.3. Interne Sicht: Überblick über die Programmteile, Datenstrukturen und Schnittstellen

In der internen Sicht auf die Programmierungsumgebung zeigt sich zunächst wieder die Zweiteilung in Spezifikations- und Synthesephase. Zu den weiteren Ausführungen ist Bild 20 zu betrachten. Die Rechtecke kennzeichnen ausführende Einheiten, die Ovale stehen für Daten; Pfeile geben den Datenfluß an.

Eine detailliertere Sicht des Spezifikationsblockes verdeutlicht, daß die vom Benutzer erstellten Bilder d.h. genauer eine Interndarstellung der Bilder, durch ein **Graphikauswerteprogramm** analysiert werden. Es wurde ein Auswerteprogramm für die Kommunikationsstruktur und für die Ablaufsteuerung erstellt. /GRAD86/ Bei der Auswertung werden z.B. isolierte Knoten oder nicht identifizierbare Symbole erkannt und entsprechende Fehlermeldungen an den Benutzer gegeben.

Die aus korrekt erstellten Bildern gewonnene Information wird in internen **Listen** gespeichert. Der Aufbau der Listen wird in Kapitel 6.1.1 beschrieben.

Das **Maskensteuerprogramm** bietet dem Benutzer Masken an, die soweit als möglich mit Informationen aus den graphischen Teilen der Spezifikation versorgt sind. Dazu werden die **Rohmasken** mit den Daten, die in den Listen enthalten sind, gefüllt.

Es werden z.B. sämtliche Namen von Prozeßbündeln, Prozeßgruppen und Einzelprozessen, die vom Auswerteprogramm in der Kommunikationsstruktur erkannt wurden, an die Maske 1 (siehe Anhang 3) übergeben.

Aus der Ablaufsteuerung werden sämtliche internen Funktionen und Operationen, sowie alle Eingabeoperationen und Ausgabeoperationen gesammelt und dem Benutzer zur weiteren Verfeinerung in Folgemasken angeboten. Ein Beispiel dafür ist Bild 19.

Durch das Maskensteuerprogramm wird sichergestellt, daß vom Benutzer eine vollständige Spezifikation erstellt wird. Es wird z.B. überprüft, ob zu sämtlichen Funktionen und Operationen wenigstens die umgangssprachliche Beschreibung (d.h. in Maske 9 Teil a.)) vorhanden ist.

Eine genaue Beschreibung des Maskensteuerprogramms findet sich in Kapitel 8.2, die Datenstruktur Masken wird in Kapitel 6.1.2 vorgestellt.

Sämtliche zu einer Spezifikation eines Prozeßsystems gehörenden Daten, d.h. die mittels Masken interaktiv vom Benutzer erfragten Informationen über die nichtgraphischen Teile der Spezifikation, werden in der Datenstruktur **Prozeßsystem** unter dem Schlüssel Prozeßsystemname gespeichert. In dieser Datenstruktur wird auch der Bezug zu den graphischen Teilen der Spezifikation hergestellt. Die Datenstruktur wird in Kapitel 6.1.3 erklärt.

Der mit 1 markierte Pfeil kennzeichnet die Schnittstelle zwischen den graphischen Teilen und den nichtgraphischen Teilen der Spezifikation. Kapitel 7.1 beschreibt diese Schnittstelle, d.h. die Funktionen, mit denen vom Maskensteuerprogramm aus, auf die Datenstruktur Listen zugegriffen wird.

Die Schnittstelle zwischen Spezifikation und Synthese bildet der mit 2 gekennzeichnete Pfeil. Kapitel 7.2 enthält die Funktionen, mit denen auf die Datenstruktur Prozeßsystem zugegriffen wird. Die Programmsynthese wird u.a. durch das Ergebnis dieser Funktionen gesteuert. Diese Funktionen finden sich unter den semantischen Aktionen aus der erweiterten PEARL-Grammatik. (Anhang 1)

Neben den eben beschriebenen Funktionen, die Informationen aus der Datenstruktur Prozeßsystem liefern, wird für die Umsetzung der Spezifikation in PEARL-Code noch zusätzliches, von der aktuellen Spezifikation unabhängiges Wissen benötigt. Dieses Wissen wird aus einer Entscheidungstabelle, die **Regeln** für die Umsetzung der Spezifikation in PEARL-Code enthält, entnommen. Die Entscheidungstabelle wird in Kapitel 6.2.1 angegeben.

Das Kernstück der Programmsynthese, die **erweiterte PEARL-Grammatik**, wird nach dem Verfahren des rekursiven Abstiegs programmiert. Auf die Grammatik wird noch genauer in Kapitel 8.1 eingegangen.

Ergebnis der Umsetzung ist ein PEARL-Programmgerüst, das intern als Baum gehalten wird.

Von einem PEARL-Programmgerüst, bzw. von halbautomatischer Programmsynthese wird gesprochen, weil nur eine syntaktisch eingeschränkte Grammatik verwendet wird. Die Grammatik enthält im wesentlichen die für diese Arbeit relevanten Sprachkonstrukte (vgl. Einleitung - Motivation). Andere Sprachkonstrukte, die z.B. für die Erzeugung eines Systemteils benötigt werden, oder Formate bei Ein-/Ausgabeeinweisungen bleiben unberücksichtigt.

In Kapitel 5.2 wurde bereits angedeutet, daß diese fehlenden Angaben ebenfalls mit Hilfe von Masken und Menüs vom Benutzer erfragt werden könnten.

6. Datenstrukturen

In diesem Kapitel werden die bei der internen Sicht auf die Programmierungsumgebung angesprochenen Datenstrukturen erklärt. Zunächst sollen die Datenstrukturen aus der Spezifikationsphase, d.h. die Listen, die Rohmasken und das Prozeßsystem beschrieben werden. Für komplexere Strukturen werden Michael Jackson Diagramme verwendet. Teilweise wird die konkrete Pascal-Repräsentation angegeben.

Im darauffolgenden Unterkapitel werden die Datenstrukturen aus der Synthesephase, d.h. die Entscheidungstabelle, die die Umsetzregeln enthält, und die Struktur des PEARL-Programmbaumes einschließlich seiner Pascal-Repräsentation, vorgestellt.

6.1. Datenstrukturen der Spezifikationsphase

6.1.1. Listen

Wie aus dem Bild 20 aus Kapitel 5.3 ersichtlich, erstellt das Graphikauswerteprogramm als Ergebnis der Auswertung Listen. Diese Listen werden jeweils für die Kommunikationsstruktur und Ablaufsteuerung kurz beschrieben.

Listen, die aus der Kommunikationsstruktur erstellt werden:

- 1.) Liste aller in der Kommunikationsstruktur vorkommenden Prozesse
- 2.) Liste aller Prozeßbündel
Zu jedem Bündel wird auf die zugehörigen Prozesse verwiesen
- 3.) Liste aller Prozeßgruppen
Zu jeder Gruppe wird auf die zugehörigen Prozesse verwiesen
- 4.) Liste aller in der Kommunikationsstruktur vorkommenden Texte

5.) Liste der Kommunikationsstruktur, die Angaben über

- die Art des Senders (Rechenprozeß,
Gerät/ Benutzer,
Technischer Prozeß)
- den Sender (Name eines Prozesses)
- die Nachricht/Nachrichten (Nachrichtennamen)
- die Art des Empfängers (wie Art des Senders)
- den/die Empfänger (Name eines Prozesses)

enthält.

Intern werden diese Listen als doppelt verkettete Listen gehalten, dazu wird die PASCAL-Notation angegeben:

```
TYPE process = RECORD
    titel : ^zahlliste;
    vorgaenger, nachfolger : ^process;
END;

zahlliste = RECORD
    zahl : integer;
    vorgaenger, nachfolger : ^zahlliste;
END;

buendel = RECORD
    nummer : integer;
    komponenten : ^process;
    vorgaenger, nachfolger : ^buendel;
END;

gruppe = RECORD
    nummer : integer;
    prozesse : ^process;
    vorgaenger, nachfolger : ^gruppe;
END;
```

```

textttabelle = RECORD
    inhalt : string;
    nummer : integer;
    vorgaenger, nachfolger : ^textttabelle;
END;

ksstruktur = RECORD
    sender, empfaenger, botschaft :
                                ^zahlliste;
    sendart, empfangart : integer;
    vorgaenger, nachfolger : ^ksstruktur;
END;

```

Die unter Punkt 5 angegebene Liste entspricht der Liste der Kommunikationspartner, die im Kapitel 4.3 bei der Gliederung der Spezifikation erwähnt wurde. Diese Liste muß damit in der Programmierumgebung nicht mehr vom Benutzer eingegeben werden, sondern wird **automatisch aus der graphischen Darstellung der Kommunikationsstruktur erstellt** und dem Benutzer angeboten.

Ein Abspeichern aller angegebenen Listen auf Datei ist ebenfalls möglich.

Listen, die aus der Ablaufsteuerung erstellt werden:

1.) Liste aller in der Ablaufsteuerung vorkommenden Texte.

(Aufbau und Form entsprechen der Liste 4 aus der Kommunikationsstruktur.)

2.) Liste aller Knoten einer Ablaufsteuerung;

ein Knoten umfaßt die Komponenten Knotenart, Knoteninhalte, Zeiger auf eine verkettete Liste Kante; die Komponente Kante enthält ihrerseits die Komponenten Kantenart, Kantenbeschriftung und einen Zeiger auf den Zielknoten. Die verkettete Liste Kante enthält alle von einem Knoten wegführenden Kanten.

Diese Informationen werden bei der Programmsynthese als Knoten- und Kantenattribute bezeichnet.

Zunächst wird die PASCAL-Notation für beide Listen angegeben. Danach wird die unter 2.) beschriebene Liste - besser wäre es, von einem Geflecht zu sprechen - mit einem Michael Jackson Diagramm beschrieben.

```

TYPE  zahlliste = RECORD
        zahl : integer;
        vorgaenger, nachfolger : ^zahlliste;
    END;

knoten = RECORD
        art : string;
        inhalt : knoteninhalt;
        kantenzeiger : ^kante;
        vorgaenger, nachfolger : ^knoten;
    END;

knoteninhalt = RECORD
        titel, text : ^zahlliste;
    END;

kante = RECORD
        art : string;
        beschriftung : ^zahlliste;
        zielknoten : ^knoten;
        vorgaenger, nachfolger : ^kante;
    END;

```

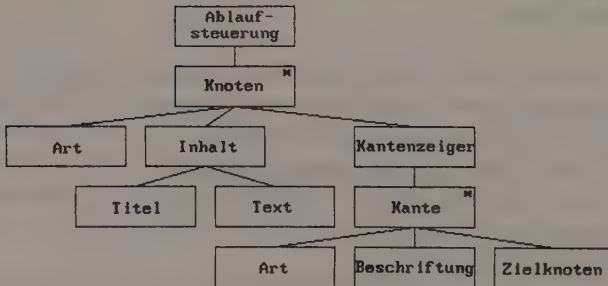


Bild 21: Michael Jackson Diagramm für die Ablaufsteuerung

6.1.2. Datenstruktur Masken

Mit Masken ist hier stets die Datenstruktur für die sogenannten Rohmasken gemeint. Die Rohmasken enthalten im wesentlichen die festen Texte einer Maske. Die Anzahl der festen Texte ist von Maske zu Maske verschieden. Die Texte können vom Benutzer nicht geändert werden.

Jede Maske erhält zu ihrer Identifizierung eine Maskennummer. Die festen Texte werden in einer Texttabelle gespeichert. In der Datenstruktur Rohmaske tauchen sie mit einer Textnummer auf. Zu jedem festen Text wird bei der Beschreibung der Rohmaske angegeben, ob dazu eine Eingabe vom Benutzer (B-Kennung) und/oder eine Information aus der Graphik, d.h. aus den Listen (G-Kennung), möglich ist.

Zu jeder Rohmaske werden die Menüoperationen, die zu dieser Maske möglich sind, angegeben.

Layout-Angaben werden vorerst nicht berücksichtigt.

Die eben beschriebene Struktur soll durch ein Michael Jackson Diagramm verdeutlicht werden.

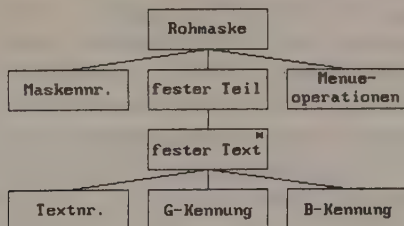


Bild 22: Michael Jackson Diagramm für Rohmasken

6.1.3. Datenstruktur Prozeßsystem

Die Rohmasken können mit Informationen aus der Graphik, d.h. aus den Listen (G-Teile), und/oder Eingaben vom Benutzer (B-Teile) gefüllt werden.

Diese G-Teile und B-Teile aller Masken bilden zusammen mit Schlüsseln zum Wiederauffinden der Information die Datenstruktur Prozeßsystem.

Die G-Teile dürfen nur aus der Graphik übernommen werden. Der Benutzer darf G-Teile weder eingeben, noch löschen noch ändern. Zu einem fixen Text in der Rohmaske sind unterschiedlich viele und beliebig lange G-Teile und B-Teile möglich. Die G-Teile sind aber bezüglich der Spezifikation eines Prozeßsystems variabel und werden deshalb nicht in den Rohmasken, sondern in der Datenstruktur Prozeßsystem in einer Tabelle gehalten. Die G-Texte, die zu einem Prozeßsystem gehören, werden in einer Tabelle, auf die ein G-Textzeiger zeigt, gehalten.

Für die B-Teile gibt es zu jeder einzelnen Maske eine Tabelle, die die entsprechenden Texte enthält (B-Textzeiger).

Zu einer Rohmaske in einer Prozeßsystemspezifikation kann es mehrere Ausprägungen geben. Ein Beispiel dafür ist die Rohmaske für die Ausgabefunktion (vgl. Bild 19). Diese Maske wird für die Spezifikation aller Ausgabefunktionen genutzt. Nach der Bearbeitung einer solchen Maske, d.h. nach der Spezifikation einer Ausgabefunktion, gibt es eine neue Version. Im Bild 19 sind die Texte 'proto' und 'byte_senden' aus der Kommunikationsstruktur bzw. aus der Ablaufsteuerung übernommen, d.h. es sind oben beschriebene G-Teile. Texte, die vom Benutzer unter a.), b.) oder c.) im Bild 19 eingegeben werden, sind B-Teile. Als Schlüssel unter Version müßte der Name der Ausgabefunktion, also 'byte_senden', eingetragen werden. Im folgenden wird die Datenstruktur Prozeßsystem mit Michael Jackson beschrieben:

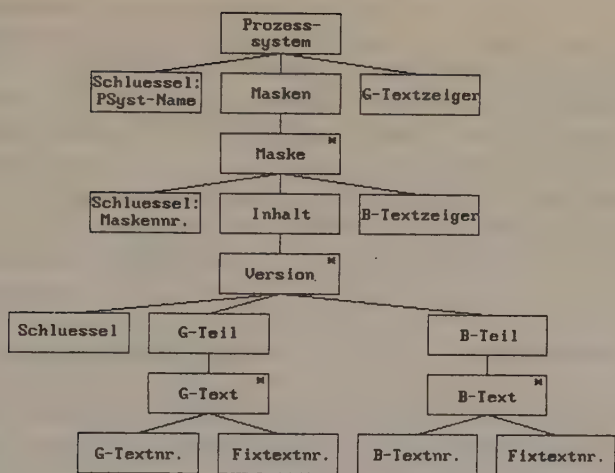


Bild 23: Michael Jackson Diagramm für Prozeßsystem

6.2. Synthese-Phase

Neben den Informationen aus der Datenstruktur Prozeßsystem liefern Umsetzregeln Hinweise, wie eine in PASS vorliegende Spezifikation in PEARL-Code umzusetzen ist.

Der erzeugte Code wird nicht zeilenweise gespeichert, sondern die Struktur des erzeugten Programms wird in einem sogenannten PEARL-Programmbaum gehalten.

Im Kapitel 6.2.1 werden oben erwähnte Umsetzregeln mit einer Entscheidungstabelle verdeutlicht.

Kapitel 6.2.2. enthält den Aufbau des PEARL-Programmbaumes.

6.2.1. Umsetzregeln

Die Umsetzregeln beziehen sich auf die Umsetzung des graphischen Objekts Knoten aus der Ablaufsteuerung in PEARL-Anweisungen.

Beim pragmatischen Umsetzen von Hand hat sich gezeigt, daß Knoten mit bestimmten Eigenschaften, den sogenannten Knotenattributen und Kantenattributen, in bestimmte PEARL-Sprachkonstrukte umgesetzt werden. Ganz allgemein bestimmen die Attribute damit die Wahl des Sprachkonstrukts. /KRAG82/, /STAR85/, /HEIL85/

Diese Vermutung wurde durch gezielte Untersuchungen bestätigt. Aufgrund dieser Untersuchungen wurden allgemeingültige Regeln für die Umsetzung aufgestellt.

In der nachfolgenden Entscheidungstabelle werden die Knotenattribute und Kantenattribute im Bedingungsteil aufgelistet. Ein J in der Regelspalte im Bedingungsteil bedeutet, daß diese Bedingung, d.h. dieses Attribut, zutrifft. Ein N zeigt an, daß dieses Attribut nicht zutrifft, ein - bedeutet, daß das Attribut keinen Einfluß auf die Entscheidung ausübt.

Ein * in der Regelspalte im Aktionsteil besagt, daß bei Vorliegen der zugehörigen Bedingungen, die mit * bezeichnete PEARL-Anweisung ausgewählt wird.

Der Aktionsteil enthält nur eine Teilmenge aller möglichen PEARL-Anweisungen. Im wesentlichen sind dies die Anweisungen, die die Kommunikation von Prozessen betreffen. Die Regeln für die Umsetzung von Internknoten wurden nicht in die Entscheidung

derungstabelle aufgenommen. Sie werden bei der Beschreibung der Programmsynthese aufgeführt und erklärt.

		R1	R2	R3	R4	R5	R6	R7	R8
Art des Knt.	Kommunikationsknt.	J	J	J	J	J	J	J	J
	Internknt.	N	N	N	N	N	N	N	N
Kantenzahl	= 1	J	J	J	J	J	J	N	N
	> 1	N	N	N	N	N	N	J	J
Art der Kante	Sende-Kante	J	N	J	N	J	N	-	-
	Empfänger-Kante	N	J	N	J	N	J	-	-
	OUI-Kante	N	N	N	N	N	N	J	N
Art des Kommunikationspartners	Rechenprozess	J	J	N	N	N	N	-	-
	Geraet/ Benutzer	N	N	J	J	N	N	-	-
	technischer Prozess	N	N	N	N	J	J	-	-
Transmit-Anweisung		N							
Receive-Anweisung			N						
Put-Anweisung				N					
Get-Anweisung					N				
Send-Anweisung						N			
Take-Anweisung							N		
Guarded-Command								N	
Guarded-Region									N

Bild 24: Regeln zur Umsetzung von Kommunikationsknoten

6.2.2. PEARL-Programmbaum

Der PEARL-Programmbaum enthält den gesamten durch die Programmsynthese erzeugten PEARL-Code zu einer gegebenen Spezifikation.

Ein PEARL-Programmbaum kann beliebig viele Moduln enthalten. Jeder Modul besteht aus einem Modulkopf mit dem Modulnamen und einem Problemteil. (Der Systemteil wird im Prototyp nicht berücksichtigt.) Der Problemteil kann aus beliebig vielen Spezifikationen oder Deklarationen von Daten, Prozeduren und Tasks bestehen. (Bei einer Deklaration wird ein neues Objekt geschaffen, bei einer Spezifikation wird ein bereits bestehendes Objekt bekannt gemacht.)

Eine Taskdeklaration umfaßt den Taskkopf und den Taskrumpf. Im Rumpf können beliebig viele lokale Daten deklariert werden. Die Anweisungen bestehen immer aus einer Marke und einem Anweisungsblock.

Die eben beschriebene Struktur wird durch ein Michael Jackson Diagramm verdeutlicht:

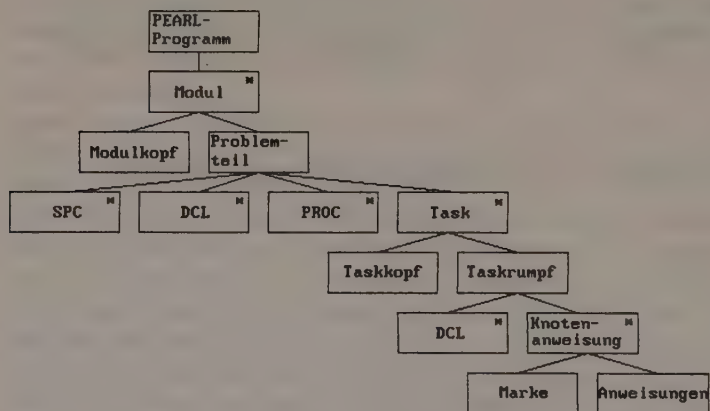


Bild 25: Michael Jackson Diagramm für den PEARL-Programmbaum

In PASCAL wird diese Struktur durch eine binären Baum verwirklicht. Jedes Bauelement enthält vier Komponenten:

```
TYPE bauelement = RECORD
    schlüssel : string;
    inhalt    : string;
    sohn      : ^bauelement;
    bruder    : ^bauelement;
END;
```

'schlüssel' soll die PEARL-Schlüsselwörter MODUL, PROBLEM, SPC, DCL, PROC, TASK und zusätzlich Markennamen enthalten. Unter 'inhalt' wird der zu dem Schlüssel gehörende PEARL-Code eingetragen.

Mit 'bruder' werden alle Knoten auf einer Ebene im Michael Jackson Diagramm verknüpft.

Mit 'sohn' wird auf den ersten (linken) Knoten auf der Ebene darunter gezeigt. Der 'sohn' eines Knotens 'Modul' ist 'Modulkopf'. Dessen 'sohn' ist leer, d.h. es gibt keine Verfeinerung. Der 'bruder' eines Knotens 'SPC' ist entweder ein weiterer Knoten 'SPC', oder, wenn nur ein Knoten 'SPC' angelegt wurde, der erste Knoten 'DCL'.

7. Schnittstellenbeschreibung

Auf die in Kapitel 6 eingeführten Datenstrukturen wird nur mittels Funktionen zugegriffen. Diese Funktionen sind gleichzeitig die Schnittstelle zwischen den einzelnen Programmstücken. Für jede Funktion wird in den nachfolgenden Kapiteln der Funktionskopf angegeben, und die Eingabe- und Rückgabeparameter beschrieben.

In Kapitel 7.1 werden nur die Funktionen angegeben, die die Information zum Füllen der Rohmasken liefern. In Kapitel 7.2 wird die Schnittstelle zwischen der Spezifikation und der Programmsynthese beschrieben. Kapitel 7.2 enthält alle Funktionen mit denen auf die Datenstruktur Prozeßsystem und mittels Informationen aus dem Maskensteuerprogramm auf die Datenstruktur Listen zugegriffen wird. Auf diese Listen kann nicht direkt von der Programmsynthese aus zugegriffen werden, da sonst die Zuordnung der Kommunikationsstruktur bzw. der einzelnen Ablaufsteuerungen zum aktuellen Prozeßsystem nicht mehr gegeben ist.

7.1. Die Schnittstelle zwischen den graphischen und nichtgraphischen Teilen der Spezifikationsmethode

Zu jeder Funktion werden die Eingabe und Rückgabeparameter beschrieben.

FUNCTION Prozeßtyp (KSname : string, Art : char) : string;

Eingabeparameter: KSname ist der Name der Kommunikationsstruktur auf die sich die Anforderung bezieht;

Art gibt die Art des gewünschten Prozeßtyps an; b für Prozeßbündel, g für Prozeßgruppe, e für Einzelprozeß

Rückgabeparameter: der Name eines Prozeßbündels, einer Prozeßgruppe oder eines Einzelprozesses; falls kein Name der gewünschten Art mehr vorliegt, wird die leere Zeichenkette als Ergebnis übergeben

```

FUNCTION Prozess (KSname : string, Prozeßtyp : char,
                 Prozeßtypname : string) : string;

```

Eingabeparameter: KSname ist der Name einer Kommunikationsstruktur;
 Prozeßtyp (b oder g) zu dem ein Prozeßname angefordert wird;
 Prozeßtypname ist der Name eines Prozeßbündels oder einer Prozeßgruppe

Rückgabeparameter: der Name eines Prozesses aus einem Prozeßbündel oder einer Prozeßgruppe;
 falls zu dem angegebenen Prozeßbündel oder der Prozeßgruppe kein Prozeß mehr vorhanden ist, wird die leere Zeichenkette übergeben

```

FUNCTION PrivBMOF (ASname : string, Art : char) : string;

```

Eingabeparameter: ASname ist der Name einer Ablaufsteuerung, die dem aktuellen Prozeß entspricht;
 Art ist die Art des gewünschten Operationentyps:
 O : interne Operation
 F : interne Funktion
 A : Ausgabefunktion
 E : Eingabeoperation

Rückgabeparameter: der Name einer Operation oder Funktion;
 falls keine Operation oder Funktion mehr vorhanden ist, wird die leere Zeichenkette übergeben;

7.2. Die Schnittstelle zwischen Spezifikation und Synthese

Die in dieser Schnittstellenbeschreibung angesprochenen Funktionen finden sich als semantische Aktionen in der PEARL-Grammatik wieder.

Um die entsprechende Information in der Datenstruktur Prozeß-

system (bzw. Listen) zu finden, werden an die Funktionen als Eingabeparameter eine Reihe von Schlüsseln übergeben. Der Eingabeparameter ist ein Zeiger auf eine verkettete Liste vom Typ Pfad.

```
TYPE Pfad = RECORD
    name : string;
    nächster : ^Pfad;
END;
```

Das erste Element dieser Liste ist der Name des aktuellen Prozeßsystems, das zweite Element, falls vorhanden, der Modulname, danach folgt der aktuelle Taskname, usw.

Das Ende der Liste wird durch nächster = NIL gekennzeichnet. (vgl. dazu Schlüssel in der Datenstruktur Prozeßsystem)

Diese Funktionen liefern entweder eine Zeichenkette, d.h. einen Namen oder das definierte Abbruchkriterium "Nullstring".

```
CONST Nullstring = "";
```

```
FUNCTION Modulname (Pfaderster : Pfadzeiger) : string;
```

Eingabeparameter: Name des Prozesssystems

Rückgabeparameter: Name einer Strukturierungseinheit oder Nullstring;

```
FUNCTION Taskname (Pfaderster : Pfadzeiger) : string;
```

Eingabeparameter: Name des Prozesssystems, Modulname

Rückgabeparameter: Name eines Prozesses in einem Prozeßbündel oder einer Prozeßgruppe;

FUNCTION Sendenachrichten_Zahl (Pfaderster : Pfadzeiger) :
Cardinal;

Eingabeparameter: Name des Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Anzahl der Sendenachrichten an andere
Prozesse (positive ganze Zahl >=0)

FUNCTION Empfangsnachrichten_Zahl (Pfaderster : Pfadzeiger) :
Cardinal;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Anzahl der Empfangsnachrichten von an-
deren Prozessen
(positive ganze Zahl >= 0)

FUNCTION Wartebereich (Pfaderster : Pfadzeiger) : Cardinal;

Eingabeparameter: Name des Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Wartebereichsgröße zum aktuellen Prozeß
(positive ganze Zahl >=0)

FUNCTION Sendenachricht (Pfaderster : Pfadzeiger) : string;

Eingabeparameter: Name des Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Name einer zu sendenden Nachricht
(Sendenachrichtenname)

FUNCTION Empfangsnachricht (Pfaderster : Pfadzeiger) :
string;

Eingabeparameter: Name des Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Name einer zu empfangenden Nachricht
(Empfangsnachrichtenname)

FUNCTION Sender (Pfaderster : Pfadzeiger) : string;

Eingabeparameter: Name des Prozesssystems, Modulname,
Taskname, Empfangsnachrichtenname

Rückgabeparameter: Absender, d.h. Name eines Prozesses
der die zu erwartende Nachricht sendet

FUNCTION Empfänger (Pfaderster : Pfadzeiger) : string;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Sendenachrichtenname

Rückgabeparameter: Empfängername, d.h. Name eines Prozesses

FUNCTION Knotenname (Pfaderster : Pfadzeiger) : string;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Name eines Knotens aus der aktuellen
Ablaufsteuerung

FUNCTION lokale_Daten (Pfaderster : Pfadzeiger) : string;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname

Rückgabeparameter: Lokale Daten aus der Spezifikation
Diese Daten werden vorerst ohne Syntax-
prüfung aus der privaten Benutzerma-
schine zum aktuellen Prozeß übernommen

FUNCTION Knotenart (Pfaderster : Pfadzeiger) : char;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Knotenname

Rückgabeparameter: I für Internknoten
K für Kommunikationsknoten

FUNCTION TO-Wert (Pfaderster : Pfadzeiger) : Cardinal;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Knotenname

Rückgabeparameter: Eintrag im Timeout-Feld aus der
Ablaufsteuerung zum aktuellen Knoten-
namen
(positive ganze Zahl ≥ 0)

FUNCTION Kantenzahl (Pfaderster : Pfadzeiger) : Cardinal;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Knotenname

Rückgabeparameter: Anzahl der Ausgangskanten zum
aktuellen Knoten aus der Ablaufsteu-
erung
(einschließlich der Timeoutkante)

FUNCTION Kantenbeschriftung (Pfaderster : Pfadzeiger) :
string;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Knotenname

Rückgabeparameter: Kantenbeschriftung unabhängig von der
Art der Kante

FUNCTION Art-des-Kommpartners (Pfaderster : Pfadzeiger) :
Char;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Sender-/Empfängername

Rückgabeparameter: R (P) für Rechenprozeß
G (B) für Gerät/ Benutzer
T (P) technischer Prozeß

FUNCTION Kantenart (Pfaderster : Pfadzeiger) : char;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Knotenname

Rückgabeparameter: S für Sendekante
E für Empfangskante
O für Operationskante
F für Funktionskante

FUNCTION Proc-Parameter (Pfaderster : Pfadzeiger) : string;

Eingabeparameter: Name eines Prozesssystems, Modulname,
Taskname, Knotenname, Prozedurname

Rückgabeparameter: Prozedurparameter zum aktuellen Pro-
zedurnamen;
die Parameter werden ohne Syntaxprüfung
übernommen

8. Beschreibung der Programmteile

In Kapitel 8.1 wird die Programmierung der erweiterten PEARL-Grammatik beschrieben, danach wird auf das Maskensteuerprogramm eingegangen.

8.1. Programmierung der erweiterten PEARL-Grammatik

Ziel der Programmsynthese ist es, aus einer vollständigen PASS-Spezifikation ein syntaktisch korrektes PEARL-Programm bzw. ein Programmgerüst zu erzeugen.

Im Anhang 1 ist eine syntaktisch eingeschränkte, aber um semantische Aktionen erweiterte PEARL-Grammatik definiert. Syntaktisch eingeschränkt bedeutet, daß nicht der gesamte AVIONIC-PEARL-Sprachvorrat, sondern nur ein Teil davon betrachtet wird. Die semantischen Aktionen geben an, wann zusätzlich welche Informationen für die Programmsynthese benötigt werden.

Diese Grammatik wird nach der Methode des rekursiven Abstiegs programmiert. Die Methode des rekursiven Abstiegs wird aber nicht wie gewohnt zur Analyse von Programmen, sondern zu deren Synthese eingesetzt.

Jedem nichtterminalen Symbol aus der Grammatik entspricht eine Prozedur, die die zu diesem Nichtterminal gehörende rechte Seite der Produktion bearbeitet. Den semantischen Aktionen entsprechen zum Teil die in 7.2 definierten Schnittstellenfunktionen; für die restlichen semantischen Aktionen gibt es für die Synthese lokale Funktionen. Die Bedeutung dieser semantischen Aktionen ist dem Anhang 1 zu entnehmen.

Trifft man bei der Abarbeitung einer Produktion auf eine semantische Aktion, wird die dazugehörige Funktion aufgerufen; trifft man auf ein Nichtterminal, so wird die entsprechende Prozedur bearbeitet. Terminale Symbole werden in das zu erzeugende PEARL-Programmgerüst übernommen.

Die nach diesem Verfahren programmierte Grammatik ist das Kernstück der Programmsynthese.

Im folgenden wird an einigen Beispielen das prinzipielle Vorgehen bei der Programmsynthese beschrieben.

Die Umsetzung in ein PEARL-Programm erfolgt ausgehend vom Startsymbol "Prozeßsystem" Top-Down gemäß der programmierten Grammatik. Die durch die Spezifikation, d.h. die Kommunikationsstruktur, definierte modulare Zerlegung in Strukturierungseinheiten wird in das zu erzeugende PEARL-Programm übernommen. Für jeden Software-Prozeß, d.h. für jeden Einzelprozeß, jede Prozeßgruppe und jedes Prozeßbündel, wird ein eigener PEARL-Modul erzeugt. Dies scheint gerechtfertigt, da Programmteile, die sehr eng zusammenarbeiten, in einen Modul gehören. Prozeßgruppen bzw. Prozeßbündel, die über gemeinsame Objekte kommunizieren, werden deshalb durch jeweils einen Modul realisiert.

Die semantische Aktion <Modulname> in der aus der Grammatik entnommenen Produktion

```
Prozeßsystem := ( <Modulname> Modul ) *;
```

hat zum einen die Aufgabe, den Namen eines Prozeßbündels, einer Prozeßgruppe oder eines Einzelprozesses zu liefern. Gleichzeitig steuert das Ergebnis dieser Aktion die Codeerzeugung, d.h. das Nichtterminal Modul (bzw. die zu Modul gehörende Prozedur) wird nur dann bearbeitet, wenn die semantische Aktion <Modulname> tatsächlich einen Namen liefert. Die Zahl der Wiederholungen beim Nichtterminal Modul ist damit abhängig vom Ergebnis der semantischen Aktion <Modulname>.

Nach der eben gegebenen Beschreibung wird zu folgender Kommunikationsstruktur je ein Modul für die Strukturierungseinheiten mit Namen leitung, proto und benutzer erzeugt.

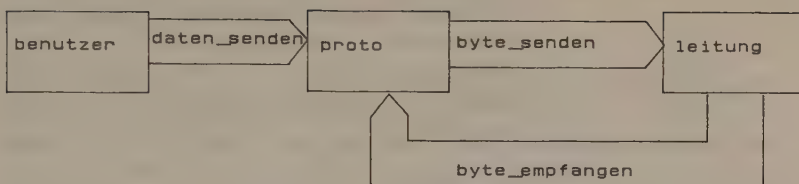


Bild 26: Beispiel einer Kommunikationsstruktur

Ein PEARL-Modul, der einem Einzelprozeß entspricht, enthält nur eine Task-Deklaration. Dagegen wird für jeden Prozeß in einer Prozeßgruppe oder einem Prozeßbündel eine eigene Task deklariert.

Da die einzelnen Prozesse einer Prozeßgruppe nicht von außen, d.h. von anderen Strukturierungseinheiten angesprochen werden dürfen, wird bei Prozeßgruppen noch eine zusätzliche Task mit dem Prozeßgruppennamen als Tasknamen erzeugt. An diese Task können von anderen Strukturierungseinheiten Nachrichten gesendet werden, diese Task darf an andere Strukturierungseinheiten Nachrichten senden. Der Wartebereich, falls vorhanden, wird ebenfalls dieser Task zugeordnet.

Allgemein besteht eine Taskdeklaration aus Taskkopf und Taskrumpf.

Die Informationen, die zur Bildung des Taskkopfes benötigt werden, werden durch semantische Aktionen zur Verfügung gestellt. Zum besseren Verständnis des nachfolgenden Beispiels werden die für die Taskdeklaration wesentlichen Produktionen aus der Grammatik (Anhang 1) angeführt:

```

Task-Declaration := Task-Identifizier ':' 'TASK'
                  Priority   Global-Resident-Attr
                  Message-Declarations ';'
                  Ablauf      ';' .
  
```

Message-Declarations :=

```
<Sendenachrichten-Zahl>
[ 'TRANSMITS' ( Transmit-Decl // ',' ) ]
<Empfangsnachrichten-Zahl>
[ 'RECEIVES' ( Receive-Decl // ',' ) ]
<Wartebereich>
('BUFFER' '(' Int-Const-Denot ')' /
 'NOBUFFER' ) .
```

Die Produktion mit dem Nichtterminal "Task-Declaration" auf der linken Seite beinhaltet die Teile zur Bildung des Taskkopfes und des Taskrumpfes. Die Angaben bis einschließlich "Message-Declarations ';' " beziehen sich auf den Taskkopf. Der Taskrumpf besteht zunächst nur aus dem Nichtterminal Ablauf, das später verfeinert wird.

Die semantische Aktion <Sendenachrichten-Zahl> liefert die Zahl der vom aktuellen Prozeß zu sendenden Nachrichten; ist diese Zahl > 0 , wird der optionale TRANSMITS-Zweig bearbeitet. In der Transmit-Decl werden dann die Nachrichtennamen und Empfängernamen eingetragen. Diese Namen werden durch Funktionen aus der Datenstruktur Prozeßsystem geliefert. Die semantische Aktion <Wartebereich> bezieht sich auf die Einträge in der zu dem aktuellen Prozeß gehörenden privaten Benutzermaschine.

Für obiges Beispiel (Bild 26) wird für den Einzelprozeß proto folgender Taskkopf erzeugt:

```
proto: TASK GLOBAL
    TRANSMITS byte_senden TO leitung
    RECEIVES daten_senden FROM benutzer,
        byte_empfangen FROM leitung
    NOBUFFER;
```

Als Ergebnis der semantischen Aktion <Wartebereich> wurde die Zahl 0 angenommen.

Aus der privaten Benutzermaschine werden auf Taskebene die lokalen Datendeklarationen übernommen. Aus der gemeinsamen Benutzermaschine werden die Daten auf Modulebene übernommen. In beiden Fällen werden die Daten nicht auf syntaktische Richtigkeit überprüft.

Auf Taskebene, d.h. im Taskrumpf, orientiert sich die Umsetzung am graphischen Objekt Knoten aus der Ablaufsteuerung. Die Knoten- und Kantenattribute bestimmen gemäß den spezifizierten Umsetzregeln (siehe 6.2.1) die Auswahl des weiter zu bearbeitenden PEARL-Sprachkonstrukts, d.h. eines Nichtterminals, für das die entsprechende Prozedur zu durchlaufen ist.

Im folgenden wird sowohl die Umsetzung der Kommunikationsknoten als auch der Internknoten am Beispiel beschrieben. Zunächst werden die für die Beispiele relevanten Produktionen aus der Grammatik angegeben:

```

Knoten-Statement := Label-Id ':'
                  Statement-Body ';' .

Label-Id :=      <Knotenname-bilden> .
Statement-Body := <Knoten-Attribute-bestimmen><Auswahl-
                  Tabelle>
                  ( Conditional-Statement /
                    Case-Statment /
                    Goto-Statement /
                    Call-Statement /
                    Get-Statement /
                    Put-Statement /
                    Transmit-Statement /
                    Receive-Statement /
                    Take-Statement /
                    Send-Statement /
                    Guarded-Region /
                    Guarded-Command /
                    Others ) .

Others :=      . /* weitere Anweisungen werden nicht
                  betrachtet */

```

```

Guarded-Region := 'GUARDED'
                  'REGION' Guards <TO-Wert> Timeout-
                              Cond ./
                  'GUARDEND' .

Guarded-Command := 'GUARDED'
                   'COMMAND' Guards <Out-Kante> Out-React
                   'GUARDEND' .

Guards := <Kantenzahl>
          ( 'GUARD'
            (Guard-Cond <Und-Kanten-
                          beschriftung> // '&' ) ';'
            Reaction ) * .

Guard-Cond := <Art-des-Kommpartners><Art-der-Kante>
              <Auswahl-Tabelle>
              (Get-Statement /
               Put-Statement /
               Take-Statement /
               Send-Statement/
               Transmit-Statement /
               Receive-Statement /
               Other-Guard-Cond ) .

Other-Guard-Cond := . /* wird nicht weiter betrachtet */

Reaction := 'REACT' <Eintrag-Benutzermaschine>
            Statement .

Timeout-Cond := 'TIMEOUT'
                'AFTER' Duration-Expression-Seven
                Reaction .

Duration-Expression-Seven :=
                        <To-Wert-bilden> .

Out-React:= 'OUTREACT' Statement .

```

Als Marke (Label-Id) wird jeweils der Knotenname aus dem graphischen Symbol übernommen. Der Kommunikationsknoten mit Namen START aus Bild 27 wird aufgrund der Knoten- und Kantenattribute und der Regel 7 aus der Entscheidungstabelle in ein GUARDED COMMAND umgesetzt. Mit Ausnahme der Out-Kante wird jede Kante aufgrund der Kantenattribute und der Regel 2 aus der Entscheidungstabelle in eine RECEIVE-Anweisung umgesetzt.

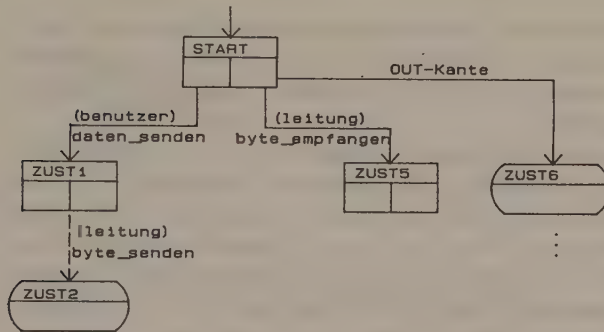


Bild 27: Auszug aus einer Ablaufsteuerung

START:

GUARDED COMMAND

GUARD RECEIVE FROM benutzer TO daten_senden
 REACT /* Ergebnis der semantischen Aktion
 <Eintrag-Benutzermaschine> */

GOTO ZUST1;

GUARD RECEIVE FROM leitung TO byte_empfangen
 REACT /* Ergebnis der semantischen Aktion
 <Eintrag-Benutzermaschine> */

GOTO ZUST5;

OUTREACT GOTO ZUST6;

GUARDEND;

Jede Eingabeoperation bzw. Ausgabefunktion muß in der privaten Benutzermaschine verfeinert werden. Wurde dabei vom Be-

nutzer PEARL-Code angegeben, so werden diese Einträge aus der Benutzermaschine nach dem Schlüsselwort REACT, allerdings ohne Syntaxprüfung, übernommen. In allen anderen Fällen wird nach REACT Kommentar eingetragen.

Bei der Programmsynthese wird im momentanen System immer ein OUTREACT-Zweig erzeugt; nach der Entscheidungstabelle wird nur aus einem Kommunikationsknoten, den u.a. eine OUT-Kante verläßt, ein GUARDED COMMAND erzeugt.

Um die Unterschiede bei der Umsetzung eines Kommunikationsknotens in ein GUARDED REGION aufzuzeigen, wird kurz auf die Semantik der beiden Konstrukte, GUARDED COMMAND und GUARDED REGION, eingegangen.

Ist bei einem GUARDED COMMAND keines der Guards ausführbar, so wird die Anweisungsfolge nach OUTREACT abgearbeitet. Nach der Syntaxdefinition für verteiltes PEARL ist die Angabe eines OUTREACT-Zweiges optional. Fehlt dieser Zweig und ist keines der Guards ausführbar, so wirkt das GUARDED COMMAND wie eine Leeranweisung. Ein Prozeß kann damit in einem GUARDED COMMAND nicht blockiert werden. Dagegen kann ein Prozeß in einem GUARDED REGION blockiert werden, wenn im Timeoutfeld keine Wartezeit eingetragen ist. Es wird dann solange gewartet, bis eines der Guards in der GUARDED REGION ausführbar wird. Üblicherweise wird eine maximal zu wartende Zeit im Timeoutfeld des betreffenden Kommunikationsknotens eingetragen. Zusätzlich ist dann noch eine Timeoutkante zu spezifizieren, die angibt, wie nach Ablauf der angegebenen Timeoutfrist weiter zu verfahren ist.

Aus der Spezifikation ist nicht zu entnehmen, welches Sprachkonstrukt gemeint ist, wenn weder ein Timeout-Eintrag noch eine Out-Kante vorhanden ist.

Bei der automatischen Programmsynthese wurde für diesen Fall die Umsetzung in ein GUARDED REGION gewählt. Eine Umsetzung in ein GUARDED COMMAND wird nur bei Angabe der Out-Kante durchgeführt. Für eine weitergehende automatische Umsetzung müßte die Spezifikation an dieser Stelle verbessert werden.

Bild 28 zeigt die Spezifikation eines Kommunikationsknotens mit einem Eintrag im Timeoutfeld:

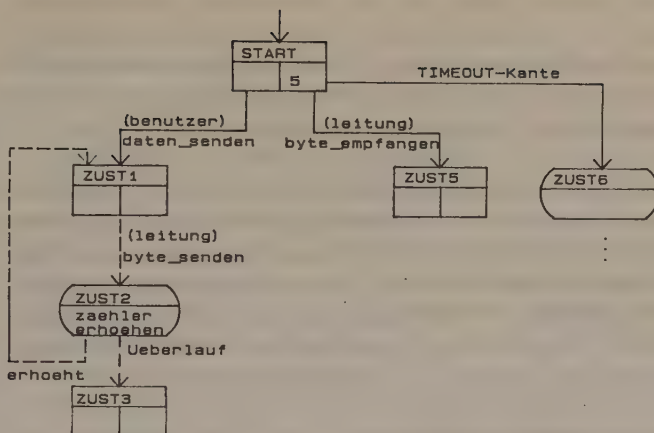


Bild 28 : Auszug aus einer Ablaufsteuerung

Der Knoten START wird aufgrund der Knoten- und Kantenattribute in ein GUARDED REGION umgesetzt. Im Beispiel wird außerdem gezeigt, welches Sprachkonstrukt nach der Regel 3 aus der Entscheidungstabelle zu wählen ist, wenn der Prozeß "benutzer" tatsächlich ein Benutzer und kein Rechenprozeß ist.

START:

GUARDED REGION

```

GUARD GET FROM benutzer TO daten_senden
  REACT /* Ergebnis der semantischen Aktion
    <Eintrag-Benutzermaschine> */
    GOTO ZUST1;
GUARD RECEIVE FROM leitung TO byte_empfangen
  REACT /* Ergebnis der semantischen Aktion
    <Eintrag-Benutzermaschine> */
    GOTO ZUST5;

```

TIMEOUT AFTER 5 SEC REACT GOTO ZUST6;

GUARDEND;

Die Möglichkeiten zur Umsetzung eines Internknotens sollen am Beispiel des Knotens ZUST2 aus Bild 28 nur kurz angesprochen werden.

Da der Schwerpunkt dieser Arbeit auf der Umsetzung von Kommunikationsknoten in entsprechende PEARL-Sprachkonstrukte lag, wird auf Internknoten nicht näher eingegangen.

Die Umsetzung des Internknotens mit dem Knotennamen ZUST2 und der internen Operation "zaehler_erhoehen" könnte z.B. nach folgender Regel erfolgen: zunächst wird eine Prozedur aufgerufen, die ein Ergebnis liefert. Abhängig von diesem Ergebnis, d.h. "zaehler ist erhoeht" oder "es hat ein Ueberlauf stattgefunden", wird in den entsprechenden Folgezustand verzweigt.

```
CALL zaehler_erhoehen (ergebnis);
CASE ergebnis
  ALT: /* erhoeht */
      GOTO ZUST1;
  ALT: /* Ueberlauf */
      GOTO ZUST3;
FIN;
```

Mit den eben an Beispielen beschriebenen Verfahren, d.h. rekursiver Abstieg, semantische Aktionen und eine Entscheidungstabelle, ist jede Situation während der Programmsynthese eindeutig bestimmt. Es gibt keine Transformationsregeln, wie sie in den Kapiteln 2.3 und 3 beschrieben wurden. Die PEARL-Codeerzeugung wird durch eine Grammatik, die nach dem Verfahren des rekursiven Abstiegs programmiert wird, gesteuert. In der Grammatik mögliche Wiederholungen, Optionen und Alternativen werden zum größten Teil durch Ergebnisse von semantischen Aktionen bestimmt. Die übrigen Entscheidungen werden aufgrund eindeutiger Regeln in der Entscheidungstabelle getroffen.

Backtracking ist an keiner Stelle notwendig.

8.2. Maskensteuerprogramm

Das Maskensteuerprogramm füllt die Datenstruktur Rohmasken soweit als möglich mit Informationen aus der Graphik und bietet die so aufbereiteten Masken dem Benutzer an.

Im Prototyp wird davon ausgegangen, daß der Benutzer zunächst die graphischen Teile der Spezifikation, d.h. die Kommunikationsstruktur und die Ablaufsteuerung für jeden Prozeß bearbeitet und sich erst anschließend den nichtgraphischen Teilen zuwendet. Diese Bearbeitungsreihenfolge wird im Maskensteuerprogramm berücksichtigt. Jedoch ist dies keine grundsätzliche Einschränkung, eine spätere Erweiterung auf eine beliebige Reihenfolge ist möglich.

Das Maskensteuerprogramm bietet dem Benutzer als erste Maske die Maske zur Definition des Prozesssystems an. Je nachdem, welche Teile der Benutzer zur weiteren Bearbeitung auswählt, wird die entsprechende Folgemaske aufgebaut. In der Menüzeile am unteren Rand des Bildschirms (siehe 5.3) werden sämtliche für die jeweils aktuelle Maske erlaubten Menüoperationen aufgelistet. Innerhalb einer Maske kann mit entsprechenden Kommandos (siehe Anhang 3) vorwärts oder rückwärts geblättert werden. Analoge Kommandos existieren für das Blättern zur Vorgängermaske. Durch Angabe einer Maskennummer oder durch Auswahl kann eine Folgemaske angewählt werden.

Das Maskensteuerprogramm stellt sicher, daß der Benutzer nur von ihm eingegebene Texte löschen, ändern oder ergänzen kann. Das Maskensteuerprogramm speichert die vom Benutzer ermittelten Informationen mit Schlüsseln in der Datenstruktur Prozeßsystem (siehe 6.1.3) ab. Die gespeicherten Daten werden auf Vollständigkeit überprüft. Beim Verlassen des Maskensteuerprogramms wird der Benutzer über fehlende Teile informiert.

Das Maskensteuerprogramm hat insgesamt vier Aufgaben zu erfüllen:

- 1.) die Rohmasken müssen mit Informationen aus der Graphik versorgt werden
- 2.) die Masken sind dem Benutzer am Bildschirm anzubieten; vom Benutzer eingetragene Daten müssen in der Datenstruktur Prozeßsystem abgelegt werden
- 3.) die abgelegten Informationen werden auf Vollständigkeit

geprüft

- 4.) das Maskensteuerprogramm liefert mittels den in 7.2 definierten Funktionen die Informationen für die Programmsynthese

9. Ausblick

Die letzten Kapitel, insbesondere die Beschreibung der Programmsynthese und der internen Sicht auf die Programmierumgebung, sollten zeigen, daß die eingangs formulierten Anforderungen von der Programmierumgebung erfüllt werden.

Die Erhöhung der Zuverlässigkeit wird durch Überprüfungen auf Spezifikationsebene und durch die programmierte Grammatik erreicht. Die Spezifikation wird hauptsächlich auf Vollständigkeit überprüft. Konsistenzprüfungen beziehen sich auf die Kommunikation von Prozessen. Untersucht wird z.B., ob es zu jeder zu sendenden Nachricht einen Empfängerprozeß gibt.

Durch Führen des Benutzers während der Spezifikation wird die Verlagerung des Entwicklungsschwerpunkts hin zur Spezifikation möglich.

Über Meinungen und Ansichten der Benutzer kann wenig ausgesagt werden, da zum Zeitpunkt der Erstellung dieser Arbeit der Prototyp noch nicht einsetzbar war.

Am bestehenden Ansatz scheint vor allem das Graphikwerkzeug für die gestellten Anforderungen ungeeignet zu sein. Es muß bei der graphischen Eingabe zumindest überprüft werden können, ob der Benutzer syntaktisch korrekte Bilder erstellt. Im Prototyp kann aufgrund des verwendeten Graphikwerkzeugs lediglich festgestellt werden, daß das Bild nicht analysierbare Elemente enthält. Ebenso sind wichtige Arbeitsschritte während der Spezifikation, wie Löschen eines Knotens oder einer Kante, oder Löschen oder Ändern einer Kantenbeschriftung nur mit viel Mühe durch den Vergleich zweier Interndarstellungen der Bilder nachzuvollziehen. Auf diese Möglichkeit wurde im Prototyp verzichtet. Diese Informationen sind aber notwendig, wenn Änderungen in der Spezifikation in bereits erzeugte und vom Benutzer ergänzte Programme übernommen werden sollen. Dazu müssen bereits vorhandene Teile im erzeugten PEARL-Programmbaum ersetzt werden können.

Da diese Informationen im Prototyp nicht zur Verfügung stehen, werden Graphiken demnach immer "neu" umgesetzt. Konsistenz zwischen Spezifikation und vom Benutzer erweiterten Programm ist damit nicht sicherzustellen, zumal dem Benutzer für die Entwicklung des Programms mit der Programmierumgebung

und Übersetzung, Binden, Laden und Ausführen unterschiedliche Systeme zur Verfügung gestellt werden. Bild 29 verdeutlicht die Rolle des Benutzers bei der Entwicklung eines Programms auf dem Arbeitsplatzrechner und dem Ausführungsrechner.

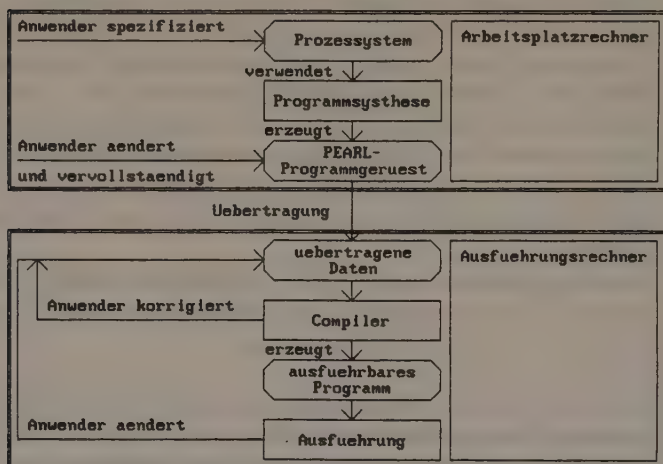


Bild 29: Arbeitsschritte bei der Entwicklung eines Programms

Insgesamt ist eine Erweiterung der Programmierumgebung in Richtung einer Software-Produktionsumgebung denkbar. Zusätzliche Prüfungen und Tests auf Spezifikationsebene können die Zuverlässigkeit der erstellten Software weiter erhöhen. An vielen Stellen wird zur Zeit während der Programmsynthese PEARL-Code ohne Syntaxprüfung übernommen. Der Anspruch, nur korrekte Programme zu erzeugen, kann damit nicht aufrechterhalten werden. Der Anschluß eines Parsers erscheint dringend erforderlich, zumal auf dem Arbeitsplatzsystem für die Programmierumgebung kein PEARL-Compiler zur Verfügung steht. Ein Parser kann auch zur Überprüfung der Benutzerergänzungen im automatisch erzeugten PEARL-Programmbaum sinnvoll eingesetzt werden. Der Benutzer kann bei den Ergänzungen im automatisch erzeugten Programmbaum mit Masken unterstützt werden. Eine

andere Möglichkeit wäre der Einsatz eines Struktureditors. Texteditoren scheiden von vorneherein aus.

Bei einer wiederholten Umsetzung, d.h. einer Korrektur oder Aktualisierung einer Spezifikation, können Teile des erzeugten PEARL-Programmbaumes wie Knoten, Tasks oder Moduln ersetzt werden. Die übrigen Teile werden unverändert belassen. Dieses Vorgehen setzt allerdings, wie bereits angesprochen, während der Programmsynthese Wissen über die in der Spezifikation erfolgten Änderungen voraus.

Auf dem erzeugten und vom Benutzer ergänzten Baum können Modulschnittstellen-Überprüfungen durchgeführt werden.

Zur Verteilung der Software-Moduln auf die zur Verfügung stehenden Hardware-Prozessoren müssen dem Benutzer Hilfsmittel und Werkzeuge angeboten werden.

Die Umsetzung der Internknoten muß noch verbessert werden. Da der Benutzer in den Internknoten die Algorithmen vorgibt, kann für Internknoten wohl kaum ein Regelsystem wie für Kommunikationsknoten entwickelt werden. Möglicherweise kann in diesen Fällen Interaktion mit dem Benutzer während der Synthese weiterhelfen.

Auch die modulare Zerlegung der Programmierungsumgebung kann für eine Erweiterung von Nutzen sein. Es erscheint auch interessant Umsetzregeln nach BASIC-PEARL anzugeben. Bei der Umsetzung nach BASIC-PEARL müssen die Nachrichten, vereinfacht ausgedrückt, in Daten mit zugehörigen synchronisierten Zugriffsoperationen abgebildet werden.

Schlußbemerkung

An dieser Stelle möchte ich mich besonders bei Herrn Dr. P. Holleczeck und Herrn Prof. Dr. H. J. Schneider für die Betreuung dieser Arbeit bedanken.

Herrn Prof. Dr. F. Hofmann danke ich für die Übernahme des Zweitgutachtens.

Meinen Kollegen danke ich für Anregungen und sorgfältiges Korrekturlesen.

Literaturverzeichnis:

- ACM85 - : "Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments", Vol. 20, Nr. 7, July 1985
- AVIO83 - : "Z80-PEARL Sprachbeschreibung (AVIONIC PEARL)", Regionales Rechenzentrum und Physikalisches Institut Universität Erlangen 1983
- BARR82 A. Barr, E. A. Feigenbaum: "The Handbook of Artificial Intelligence", Volume II, HeurisTech Press Stanford California, 1982
- BAUE G. Bauer: "Erstellung des Maskensteuerprogramms", Studienarbeit am IMMD II, Universität Erlangen, (in Arbeit)
- BAUE76 F. L. Bauer, J. Eickel (Hrsg.): "Compiler Construction An Advanced Course", Springer Verlag 1976
- BIBE80 W. Bibel: "Syntax-Directed, Semantics-Supported Program Syntheses", Artificial Intelligence 14 (1980) p.243-261
- BIER83 A. W. Biermann, G. Guiho: "Computer Program Synthesis Methodologies", Proceedings of the NATO Advanced Study Institute, D. Reidel Publishing Company 1983
- EPOS83a - : "EPOS-Einführung", Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart und GPP Gesellschaft für Prozeßrechnerprogrammierung München, (Hrsg.: R. Lauber) 4. Auflage 1983
- EPOS83b - : "EPOS-Kurzbeschreibung", Institut für Regelungstechnik und Prozeßautomatisierung der Universität Stuttgart und GPP, 1983
- EPOS84a - : "EPOS-Handbuch Teil 4: EPOS-S", GPP München, Juli 84
- EPOS84b - : "EPOS-Info", Nr. 5 Juni 1984, S.36-40

- EPOS85a - : "EPOS-Info", Nr. 8 Mai 1985, S.25-35
- EPOS85b - : "EPOS-Handbuch Die Codeumsetzung", GPP München 1985
- FLEI83 A. Fleischmann et al.: "Synchronisation und Kommunikation verteilter Automatisierungsprogramme", Angewandte Informatik, Wiesbaden 1983
- FLEI84 A. Fleischmann: "Ein Konzept zur Darstellung und Realisierung von verteilten Prozeßautomatisierungssystemen", Dissertation Universität Erlangen 1984
- GRAD86 W. Gradl: "Rechnergestützte Analyse von grafischen Elementen der Spezifikationsmethode PASS", Studienarbeit am IMMD II, Universität Erlangen 1986
- HAUS81 H. L. Hausen, M. Müllerburg: "Software-Produktionsumgebungen: Entwicklungsstand und Trends", Informatik-Fachberichte 43, (Hrsg. G. Goos) 1981
- HEIL85 F. Heilmeier: "Erfassung von Telefongesprächsdaten mit einem verteilten System von Mikrorechnern", Diplomarbeit am IMMD IV, Universität Erlangen 1985
- HERM L. Hermann: "Programmierung der erweiterten PEARL-Grammatik nach dem Verfahren des rekursiven Abstiegs", Studienarbeit am IMMD II Universität Erlangen (in Arbeit)
- HÜNK81 H. Hünke (Hrsg.): "Software Engineering Environments", North-Holland Publishing Company 1981
- KIMM79 R. Kimm, W. Koch, W. Simonsmeier, F. Tontsch: "Einführung in Software Engineering", Walter de Gruyter 1979
- KLEB83 G. Klebes: "Entwicklung eines botschaftsorientierten Synchronisationsverfahrens für parallele Prozesse und seine Realisierung im Rahmen eines Realzeitprogrammiersystems", Diplomarbeit am IMMD II Universität Erlangen 1983

- KRAG82 G. Kragl: "Entwicklung eines Sicherungsverfahrens für ein Filetransferprotokoll zwischen Kleinrechnern und der CYBER des RRZE", Diplomarbeit am IMMD IV Universität Erlangen 1982
- KRAG84 G. Kragl: "Eine Spezifikationsmethode für verteilte Systeme", Vorträge zur PEARL-Tagung 84, (herausgegeben vom PEARL-Verein), S. 80 - 88
- KRAL85 A. Krall: "PEG - Ein Programmierungsgenerator", Informatik Fachberichte 108, S. 488 - 499, Springer Verlag 1985
- KUMM83 R. Kummer: "Entwicklung von Betriebssystembausteinen für Guarded Regions und Botschaftsoperationen im Rahmen eines Realzeitprogrammiersystems", Diplomarbeit am IMMD IV der Universität Erlangen 1983
- LUDE85 J. Ludewig, M. Glinz, H. Matheis: "Software-Spezifikation durch halbformale, anschauliche Modelle", Informatik Fachberichte 108, S. 192 - 204, Springer Verlag 1985
- MANN79 Z. Manna, R. Waldinger: "Synthesis: Dreams => Programs", IEEE Transactions on Software Engineering, Vol SE-5, No 4, July 1979, p.294 - 327
- MANN80 Z. Manna, R. Waldinger: "A Deductive Approach to Program Synthesis", ACM TOPLAS, Vol 2, No 1, Jan. 1980, p. 90 - 121
- PERS85 G. Persch: "Phasenübergreifende Software-Entwicklung mit Hilfe eines Transformations-Expertensystems", Informatik Fachberichte 108, S. 143 - 155, Springer Verlag 1985
- RECH85 P. Rechenberg, H. Mössenböck: "Ein Compiler-Generator für Mikrocomputer", Carl Hanser Verlag 1985
- RETT84 J. Retti et al.: "Artificial Intelligence", Teubner 1984

- SCHN75 H. J. Schneider: "Compiler - Aufbau und Arbeitsweise",
Berlin: de Gruyter, 1975
- STAR85 U. Starke: "Korrektur von Schwankungen in Zeitzeigen
kernphysikalischer Koinzidenzelektronik mit Hilfe eines
Mikrocomputers", Diplomarbeit am Physikalischen Institut
Abt. III, Universität Erlangen 1985
- TRU085 K. Truöl, U. Viebeg: "Strukturierter Programmentwurf",
GMD-Bericht Nr. 150, Oldenbourg Verlag 1985
- ZP86 - : "Zeichenprogramm Benutzer - Handbuch", Version 3.0,
S.E.P.P. Röttenbach, 1985

Im folgenden wird eine syntaktisch eingeschränkte, aber um semantische Aktionen erweiterte PEARL-Grammatik definiert. Eingeschränkt bedeutet in diesem Zusammenhang, daß nur ein Teil des gesamten AVIONIC-PEARL Sprachvorrats betrachtet wird. So gibt es z. B. keine Möglichkeit, einen Systemteil zu definieren und auf Problementeilebene können weder Längen- noch Genauigkeitsangaben spezifiziert werden. In der Grammatik sollen solche Symbole, die nicht weiter bearbeitet werden sollen eine leere rechte Seite in der Produktion haben.

Die Grammatik wird jedoch so flexibel definiert, daß ein späterer Ausbau auf den gesamten AVIONIC-PEARL Sprachvorrat ohne wesentliche Änderung der bis dahin bestehenden Grammatik (und Synthese) möglich ist, d.h. es werden lediglich neue Produktionen, neue rechte Seiten (und Aktionen) hinzugefügt. Die Erweiterung um semantische Aktionen ist für die PEARL-Programm Synthese notwendig. Damit wird beschrieben, wann welche Aktionen bei der Bearbeitung eines bestimmten Symbols während der Synthese durchgeführt werden sollen.

Die Syntax der folgenden Grammatik orientiert sich an der EBNF-ähnlichen Syntax-Beschreibung von AVIONIC-PEARL.

Beschreibung der Metasprache:

/ zur Trennung von Alternativen
[] Optionen
* der Term vor dem Stern muß mindestens einmal vorkommen
\$ der Term vor dem \$-Zeichen kann beliebig oft vorkommen oder vollständig fehlen
() runde Klammern schließen mehrere Symbole zu einer Einheit, die im folgenden als Ganzes zu betrachten ist
// das dem //-Zeichen nachfolgende terminale Symbol trennt mehrfach vorkommende Terme, wobei der Term vor dem //-Zeichen steht
/* */ Kommentar

< > semantische Aktionen d.h. Namen von semantischen Aktionen; (die semantischen Aktionen werden im Anschluß an die Produktionen genauer beschrieben)

Eine semantische Aktion in der Grammatik bezieht sich auf das ihr folgende Symbol; eine semantische Aktion muß vor der Bearbeitung des nachfolgenden Symbols durchgeführt werden, denn sie liefert Hinweise für die Bearbeitung dieses Symbols.

Steht eine Aktion alleine auf der rechten Seite einer Produktion, so ist das entsprechende Nonterminal der linken Seite durch das Ergebnis der semantische Aktion zu ersetzen.

In der Grammmatik wurde bewußt auf eine Unterscheidung zwischen semantischen Aktionen, Attributen und Kontextbedingungen verzichtet, da letztlich jede Aktion die PEARL-Code Erzeugung beeinflusst.

Grammatik:

Schlüsselwörter werden in Großbuchstaben angegeben und in Hochkommas ('TERMINAL') eingeschlossen. Die übrigen terminalen Symbole wie , ; : werden ebenfalls in Hochkommas angegeben.

Die Produktionen werden in folgender Form angegeben:

```
Produktionsname := rechte Seite der Produktion .
                  /* Punkt als Abschlußzeichen */
```

Startsymbol sei das Nonterminal Prozeßsystem.

```
Prozeßsystem := ( <Modulname> Modul ) * .
```

```
Modul          :=      'MODULE' Modul-identifizier ';'
                        Systemteil
                        'PROBLEM' ';'
                        Declarations-and-Specifications
                        'MODEND' ';' .
```

```
Modul-identifizier := <Modulname-bilden> .
```

```
Systemteil      := .          /* wird nicht weiter betrachtet */
```

```
Declarations-and-Specifications :=
                        Length-Global-Module-Id-Declaration

                        (<Taskname> Task-Declaration/
                        Procedure-Declaration) $.
```

```
Length-Global-Modul-Id-Declaration
                        := .      /* wird nicht weiter betrachtet */
```

```
Task-Declaration := Task-Identifizier ':' 'TASK'
                        Priority      Global-Resident-Attr
                        Message-Declarations ';'
                        Ablauf        ';' .
```

```

Task-Identifizier := <Taskname-bilden> .

Priority := . /* wird nicht weiter betrachtet */

Global-Resident-Attr := Resident 'GLOBAL'.
/* GLOBAL muß im momentanen PEARL-System
immer angegeben werden */
Resident := . /* wird nicht weiter betrachtet */

Message-Declarations :=
    <Sendenachrichten-Zahl>
    [ 'TRANSMITS' ( Transmit-Decl // ',' ) ]
    <Empfangsnachrichten-Zahl>
    [ 'RECEIVES' ( Receive-Decl // ',' ) ]
    <Wartebereich>
    ('BUFFER' '(' Int-Const-Denot ')' /
    'NOBUFFER') .

Int-Const-Denot := <Wartebereich-Größe> .

Transmit-Decl := Sendenachrichtenliste
Message-Parameters
'TO'
Empfängerliste .

Receive-Decl := Empfangsnachrichtenliste
Message-Parameters
'FROM' Senderliste .

Sendenachrichtenliste := <Sendenachricht> .

Empfangsnachrichtenliste := <Empfangsnachricht> .

Empfängerliste := <Empfängerzahl>
(<Empfänger> /
( '(' <Empfänger> // ',' ' ' ) ) ) .

Senderliste := <Senderzahl>
(<Sender> /
( '(' <Sender> // ',' ' ' ) ) ) .

```

```

Message-Parameters := . /* wird nicht weiter betrachtet */

Ablauf := (Local-Identifier-Declaration ';' ) $
          ( <Knotenname> Knoten-Statement ) $
          'END' .

Local-Identifier-Declaration := < lokale-Daten > .

Knoten-Statement := Label-Id ':'
                   Statement-Body ';' .

Label-Id := <Knotenname-bilden> .

Statement-Body := <Knoten-Attribute-bestimmen><Auswahl-Tabelle>
                  ( Conditional-Statement /
                    Case-Statement /
                    Goto-Statement /
                    Call-Statement /
                    Get-Statement /
                    Put-Statement /
                    Transmit-Statement /
                    Receive-Statement /
                    Take-Statement /
                    Send-Statement /
                    Guarded-Region /
                    Guarded-Command /
                    Others ) .

Others := . /* weitere Anweisungen werden nicht
            betrachtet */

Guarded-Region := 'GUARDED'
                  'REGION' Guards <TO-Wert> Timeout-Cond /
                  'GUARDEND' .

Guarded-Command := 'GUARDED'
                   'COMMAND' Guards <Out-Kante> Out-React
                   'GUARDEND' .

```

```

Guards :=          <Kantenzahl>
                  ( 'GUARD'
                    (Guard-Cond <Und-Kantenbeschriftung> // '&' ) ';'
                    Reaction ) * .

Guard-Cond         :=<Art-des-Kommpartners><Art-der-Kante>
                    <Auswahl-Tabelle>
                    (Get-Statement /
                    Put-Statement /
                    Take-Statement /
                    Send-Statement/
                    Transmit-Statement /
                    Receive-Statement /
                    Other-Guard-Cond ) .

Other-Guard-Cond := .      /* wird nicht weiter betrachtet */

Reaction :=         'REACT' <Eintrag-Benutzermaschine>
                    Statement .

Timeout-Cond :=     'TIMEOUT'
                    'AFTER' Duration-Expression-Seven
                    Reaction .

Duration-Expression-Seven :=
                    <To-Wert-bilden> .

Out-React:=         'OUTREACT' Statement .

Receive-Statement := 'RECEIVE'
                    'FROM' Task-Id
                    'TO' Message-Id .

Task-Id :=          <Kantenbeschriftung-Prozeß> .

Message-Id :=       <Kantenbeschriftung-Nachricht> .

```

```

Transmit-Statement :=
    'TRANSMIT'
    'FROM' Message-Id
    'TO' Task-Id .

Get-Statement :=
    'GET'
    'FROM' File-Element
    'TO' One-Symbol-Or-List
    Standard-Format-List .

Put-Statment :=
    'PUT'
    'FROM' Symbol-Or-Charconstant-List
    'TO' File-Element
    Standard-Format-List .

File-Element :=
    <Kantenbeschriftung-Prozeß> .

Standard-Format-List := . . /* die Formatliste
                               wird nicht weiter betrachtet */

One-Symbol-Or-List := <Kantenbeschriftung-Nachricht> .

Symbol-Or-Charconstant-List := <Kantenbeschriftung-Nachricht> .

Take-Statement :=
    'TAKE'
    'FROM' Device-Element
    'TO' Symbol
    P-Option .

Send-Statement :=
    'SEND'
    'FROM' Symbol
    'TO' Device-Element
    P-Option .

Device-Element :=
    <Kantenbeschriftung-Prozeß> .

Symbol :=
    <Kantenbeschriftung-Nachricht> .

P-Option :=
    . /* die Formatliste
        wird nicht weiter betrachtet */

```

Conditional-Statement :=

'IF' Bit-One-Expr-Seven
'THEN' Statement *
'ELSE' Statement *
'FIN' .

Case-Statement := 'CASE' Integer-Expr-Seven

('ALT' Statement \$) *
'OUT' Statement *
'FIN' .

Statement := Goto-Statement .

Goto-Statement := 'GOTO' Zielknotenname-Identifizier .

Zielknotenname-Identifizier := <Zielknotenname> .

Bit-One-Expr-Seven := <Kantenbeschriftung> .

Integer-Expr-Seven := <Kantenbeschriftung> .

Call-Statement := 'CALL' Prozedur-Identifizier
Expression-Seven-Pack .

Prozedur-Identifizier := <Prozedurname> .

Expression-Seven-Pack := <Prozedurparameter> .

Procedure-Declaration := . /* wird nicht weiter betrachtet */

Beschreibung der semantischen Aktionen:

<Modulname>

Liefert den Namen eines Prozeßbündels, einer Prozeßgruppe oder eines Einzelprozesses.

Ein Prozeßsystem besteht aus mindestens einem Modul, im Prototyp wird für jedes Prozeßbündel, jede Prozeßgruppe und jeden Einzelprozesse ein eigener Modul erzeugt.

Die Produktion Modul ist nur dann aufzurufen, wenn ein Name existiert.

<Modulname-bilden>

Modul-Identifizier ist durch einen Modulnamen, der aus dem Ergebnis von <Modulname> gebildet wird, zu ersetzen.

<Taskname>

Liefert den Namen eines Prozesses im aktuellen Prozeßbündel bzw. in der aktuellen Prozeßgruppe.

Die Produktion Task-Declaration ist aufzurufen wenn ein Name existiert. Die Zahl der Task-Deklarationen ist abhängig von der Zahl der Prozesse im Prozeßbündel bzw. in der Prozeßgruppe.

Einem Einzelprozeß entspricht eine Task, der Modulname wird als Taskname übernommen.

<Taskname-bilden>

Der Task-Identifizier ist durch einen Tasknamen der aus dem Ergebnis von <Taskname> gebildet wird zu ersetzen.

<Sendenachrichten-Zahl>

Liefert die Zahl der Sendenachrichten für den aktuellen Prozeß;

wenn die Zahl > 0 ist, dann wird der Transmit-Zweig bearbeitet, andernfalls übersprungen.

<Empfangsnachrichten-Zahl>

Liefert die Zahl der Empfangsnachrichten für den aktuellen Prozeß;

wenn die Zahl > 0 ist, wird der Receives-Zweig bearbeitet.

<Wartebereich>

Liefert die Größe des Wartebereichs;
wenn die Zahl > 0 ist, wird der Buffer-Zweig bearbeitet;
ist die Zahl $= 0$, wird der Nobuffer-Zweig bearbeitet.

<Wartebereich-Größe>

Int-Const-Denot wird durch das Ergebnis von <Wartebereich> ersetzt.

<Senderzahl>

Liefert die Zahl der Sender, von denen eine bestimmte Nachricht erwartet wird.

<Empfängerzahl>

Liefert die Zahl der Empfänger, an die eine bestimmte Nachricht gesendet wird.

<Nachricht> <Sender> <Empfänger>

Liefern einen Nachrichtennamen bzw. Sendernamen bzw. Empfänger-
namen.

<lokale-Daten>

Liefert aus der privaten Benutzermaschine zum aktuellen Prozeß die lokalen Daten; die Daten werden ohne Syntaxprüfung übernommen.

<Knotenname>

Liefert den Namen des aktuellen Knotens;
solange ein noch nicht bearbeiteter Knoten zum aktuellen Prozeß vorhanden ist, bearbeite Knoten-Statement.

<Knotenname-bilden>

Die Marke ist durch einen Namen der aus dem Ergebnis von <Knotenname> gebildet wird zu ersetzen.

<Knoten-Attribute-bestimmen>

Liefert die Knotenattribute <Art-des-Knotens>, <Art-der-Kante>, <Kantenzahl>, <Kantenbeschriftung>, <To-Wert>, <Art-des-Kommpartners>, <Out-Kante-vorhanden>.

<Auswahl-Tabelle>

Abhängig von den Knoten-Attributen wird eines der nachfolgenden statements aus der Entscheidungstabelle ausgewählt und weiter bearbeitet; die Attribute werden entsprechend weitergereicht; zur vollständigen Dokumentation wird jedoch explizit für jede Produktion angegeben, welche Attribute benötigt werden.

<Art-des-Knotens>

Kommunikationsknoten und Internknoten;

Achtung: bei Internknoten ist immer zuerst in der Benutzermaschine zu suchen, ob dort PEARL-Code definiert ist. Ist dies der Fall, wird dieser Code übernommen.

<To-Wert>

Liefert den Eintrag im Timeout-Feld.

<To-Wert-bilden>

Ergebnis von <To-Wert> für TIMEOUT-Zweig verwenden.

<Kantenzahl>

Liefert die Zahl der Kanten (ohne Out- und Timeout-Kante), die die Anzahl der Guards bestimmt.

<Art-des-Kommpartners>

Liefert die Art des Kommunikationspartners; abhängig von der Art des Kommunikationspartners und der Art der Kante wird eines der nachfolgenden Statements ausgewählt und weiter bearbeitet.

<Out-Kante-vorhanden>

Zeigt an ob eine Out- oder λ -Kante vorhanden ist.

<Out-Kante>

Outreact wird weiter bearbeitet.

<Zielknoten>

Liefert den Namen des Zielknotens; der nachfolgende Identifier wird durch diesen Knoten ersetzt.

<Kantenbeschriftung-Prozeß>

Liefert den Namen des Prozesses an den eine Nachricht gesendet wird, bzw. von dem eine Nachricht empfangen wird.

<Kantenbeschriftung-Nachricht>

Liefert den Namen der Nachricht, die gesendet oder empfangen wird.

<Und-Kantenbeschriftung>

Prüft, ob eine Und-Kantenbeschriftung vorliegt.

<Kantenbeschriftung>

Liefert die Kantenbeschriftung von Kanten, d.h. die Kantenbeschriftung für Sendekanten, Empfangskanten, Funktionskanten und Operationskanten.

<Prozedurname>

Liefert den Namen einer Prozedur aus der Benutzermaschine.

<Prozedurparameter>

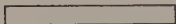
Liefert die Prozedurparameter zum <Prozedurnamen>.

Anhang 2

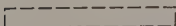
Im folgenden werden die für die **Kommunikationsstruktur** vordefinierten Symbole aufgelistet.

Jedes der Symbole darf beliebig vergrößert oder verkleinert und um 90 Grad gedreht dargestellt werden. Als Kanten sind Polygonzüge zu zeichnen, die eine beliebige geradzahlige Anzahl von Vektoren enthalten. Die Kanten müssen das jeweilige Knotensymbol berühren. Die Beschriftung der Knoten und Kanten hat innerhalb der Symbole zu erfolgen und zwar zeilenweise mit einem ';' als Trennzeichen.

Symbole fuer Rechenprozesse



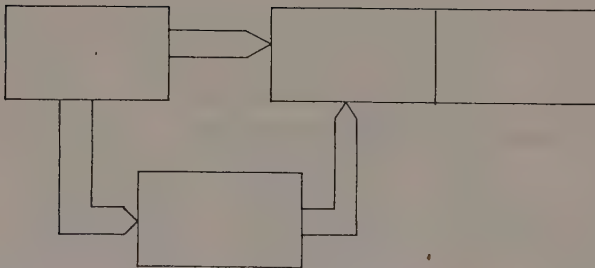
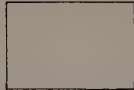
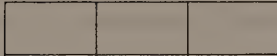
Symbole fuer technische Prozesse



Symbole fuer Gerate und Benutzer



Symbole fuer Prozessbuenchel und -gruppen



Im folgenden werden die für die **Ablaufsteuerung** vordefinierten Symbole angegeben.

Anders als bei der Kommunikationsstruktur darf keines der unten angegebenen Symbole vergrößert oder verkleinert werden. Außer dem Symbol "pfeil" darf auch keines der Symbole gedreht werden.

Das Symbol "Konnektor" wird zur Unterbrechung langer Kanten und zur übersichtlichen Darstellung mehrseitiger Bilder verwendet.

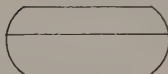
Mit dem Symbol "Makro" können mehrfach vorkommende Teilstücke aus der Ablaufsteuerung zusammengefaßt werden.

Die Beschriftung hat innerhalb der Symbole zu erfolgen. Für jede Kantenbeschriftung muß durch zusätzliche Geraden die Beziehung zur Kante hergestellt werden. (Dieser Bezug ist in Schicht 2 der Bilder herzustellen).

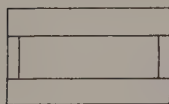
Kommunikationsknoten



Internknoten



Makroknoten



Pfeil ✓

Konnektor



Anhang 3

Anhang 3 enthält den Aufbau sämtlicher Masken, mit deren Hilfe der Benutzer durch die nichtgraphischen Teile der Spezifikationsmethode PASS geführt wird. Zunächst wird der allgemeine Aufbau einer Maske angegeben:

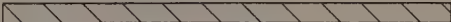
Zeile fuer Fehlermeldungen	
Maskenfenster	Auswahlspalte
Menue-Zeile	

Das Maskenfenster enthält im einzelnen folgende Angaben:

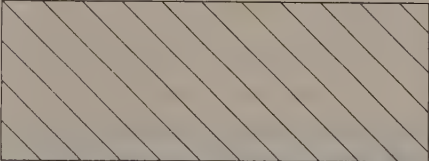
In jeder Maske wird eine Nummer und eine Bezeichnung angegeben. Zweistellige Maskennummern wie 1.1 und 1.2 kennzeichnen inhaltlich zusammengehörende Masken, die aus organisatorischen Gründen (Bildschirmgröße) nicht in einer Maske dargestellt werden konnten. Die Masken-Bezeichnung informiert kurz über den Maskeninhalt. Auf diese Bezeichnung folgen im oberen Drittel der Maske Angaben für den Benutzer über den bereits durchlaufenen Weg in der Spezifikation. Beispiele dafür sind in Maske 2 die Texte II.1.1 Typen von Prozeßbündeln und Prozeßbündelname. Diese Texte informieren den Benutzer darüber, daß in der aktuellen Maske ein Prozeßbündel mit dem angegebenen Namen weiter spezifiziert werden soll.

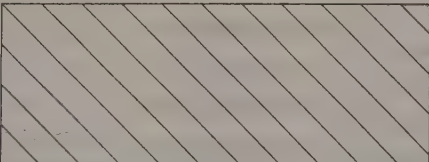
Texte, die aus der Graphik übernommen werden, sind unterstrichen. Die Stellen, an denen der Benutzer Text eintragen darf, sind mit schraffierten Kästen gekennzeichnet. Die übrigen Texte sind die festen Texte einer Maske.

In der Auswahltspalte sind ebenfalls mit Kästen die Stellen markiert, an denen der Benutzer Teile zur weiteren Bearbeitung auswählen kann.
Die Menüzeile enthält sämtliche für die Maske erlaubten Kommandos.

Zeile fuer Fehlermeldungen	
Maske 1.1: Gliederungsschema der Spezifikation	
I.	Kommunikationsstruktur
I.1	Kommunikation zwischen Typen von Strukturierungseinheiten
	
II.	Definition des Prozesssystems
II.1	Typen
II.1.1	Typen von Prozessbuendeln
	<u>erster Name aus I.1</u>
	<u>zweiter Name aus I.1</u>
	...
II.1.2	Typen von Prozessgruppen
	<u>Namen aus I.1</u>
II.1.3	Typen von Einzelprozessen
	<u>Namen aus I.1</u>
Auswahl Eintragen Beenden Vorwaerts Blaettern	

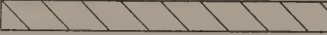
Zeile fuer Fehlermeldungen	
Maske 1.2: Gliederungsschema der Spezifikation	
II.2	Deklaration von Prozessen
II.2.1	Deklaration von Prozessbuendeln
	<u>Namen aus Maske 1.1 (II.1.1)</u>
II.2.2	Deklaration von Prozessgruppen
	<u>Namen aus Maske 1.1 (II.1.2)</u>
II.2.3	Deklaration von Einzelprozessen
	<u>Namen aus Maske 1.1 (II.1.3)</u>
Auswahl Beenden Vorwaerts Blaettern	

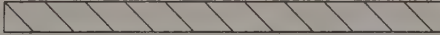
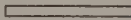
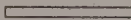
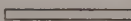
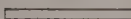
Zeile fuer Fehlermeldungen	
Maske 2: Typen von Prozessbuendeln II.1.1 Typen von Prozessbuendeln <u>Prozessbuendelname</u> A. Prozesse <u>erster Prozessname</u> <u>zweiter Prozessname</u> ... B. gemeinsame Benutzermaschine 	 ...
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts	

Zeile fuer Fehlermeldungen	
Maske 3: Typen von Prozessgruppen II.1.2 Typen von Prozessgruppen <u>Prozessgruppenname</u> A. Prozesse <u>erster Prozessname</u> <u>zweiter Prozessname</u> ... B. gemeinsame Benutzermaschine 	 ...
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts	

Zeile fuer Fehlermeldungen	
Maske 4.1: Prozess aus einem Prozessbuen-del II.1.1 Typen von Prozessbuen-deln <u>Prozessbuen-delname</u> <u>Prozessname</u> umgangssprachliche Prozessbeschreibung 1. Kommunikationsmaschine Wartebereich: 2. Ablaufsteuerung	
Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 4.2: Prozess aus einem Prozessbuen-del II.1.1 Typen von Prozessbuen-deln <u>Prozessbuen-delname</u> <u>Prozessname</u> 3. private Benutzermaschine 3.1 Ausgabefunktionen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.2 Eingabeoperationen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.3 interne Operationen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.4 interne Funktionen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.5 lokale Daten	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zelle fuer Fehlermeldungen	
Maske 5.1: Prozess aus einer Prozessgruppe II.1.1 Typen von Prozessgruppen <u>Prozessgruppenname</u> <u>Prozessname</u> umgangssprachliche Prozessbeschreibung 	
1. Kommunikationsmaschine Wartebereich: 	
2. Ablaufsteuerung 	
Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zelle fuer Fehlermeldungen	
Maske 5.2: Prozess aus einer Prozessgruppe II.1.1 Typen von Prozessgruppen <u>Prozessgruppenname</u> <u>Prozessname</u> 3. private Benutzermaschine 3.1 Ausgabefunktionen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.2 Eingabeoperationen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.3 interne Operationen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.4 interne Funktionen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.5 lokale Daten 	   
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 6.2: Einzelprozess II.1.1 Typen von Einzelprozessen <u>Prozessname</u> 3. private Benutzermaschine 3.1 Ausgabefunktionen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.2 Eingabeoperationen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.3 interne Operationen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.4 interne Funktionen <u>Namen aus 2. (Ablaufsteuerung)</u> 3.5 lokale Daten 	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 6.1: Einzelprozess II.1.1 Typen von Einzelprozessen <u>Prozessname</u> ungangssprachliche Prozessbeschreibung 	
1. Kommunikationsmaschine Wartebereich:	
2. Ablaufsteuerung	
Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 7: Private Benutzermaschine II.1.1 Typen von Prozessgruppen <u>Prozessgruppenname</u> <u>Prozessname</u> Ausgabefunktionen: <u>Name aus Maske 5</u>	
a. umgangssprachliche Beschreibung	
b. Pseudocode	
c. PEARL-Code	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 8: Private Benutzermaschine II.1.1 Typen von Prozessbuendeln <u>Prozessbuendelname</u> <u>Prozessname</u> Ausgabefunktionen: <u>Name aus Maske 4</u>	
a. umgangssprachliche Beschreibung	
b. Pseudocode	
c. PEARL-Code	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 9: Private Benutzermaschine II.1.1 Typen von Einzelprozessen <u>Prozessname</u> Ausgabefunktionen: <u>Name aus Maske 6</u> a. umgangssprachliche Beschreibung b. Pseudocode c. PEARL-Code 	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 10: gemeinsame Benutzermaschine II.1.1 Typen von Prozessgruppen <u>Prozessgruppenname</u> <u>Prozessname</u> Name einer Funktion aus Maske 2 a. umgangssprachliche Beschreibung b. Pseudocode c. PEARL-Code 	
Auswahl Eintragen Beenden Vorwaerts Rueckwaerts Blaettern	

Zeile fuer Fehlermeldungen	
Maske 11: Deklaration II.2.1 Deklaration von Prozessbuendeln <u>Name aus Maske 1.2</u> konkrete Prozessnamen: 	
Eintragen Beenden Vorwaerts Rueckwaerts Blattern	

Stichwortverzeichnis

Ablaufsteuerung	55, Anhang 2-3
Ausgabefunktion	57
Automatische Codegenerierung	28
Automatische Programmsynthese	42
Automatische Programmtransformation	42
Automatische Umsetzung	1
Automatische Verifikation	42
Automatisches Programmieren	40
Bedingte Verzweigung	32
Codeerzeugung	2
Codeselektion	27
Computerunterstütztes Programmieren	40
Eingabeoperation	57
Einzelprozeß	53
Elementaroperation	18, 29
Entscheidungstabelle	11, 17, 93, 95
Entwurfsobjekt	20, 29
EPOS	8
EPOS-A	10, 17, 24
EPOS-C	11, 27
EPOS-D	10, 17, 24
EPOS-M	10, 25
EPOS-P	10, 25
EPOS-R	10, 11
EPOS-S	10, 17
Erweiterte PEARL-Grammatik	68, Anhang 1-1
Erweiterte PEARL-Grammatik, Programmierung	87
Gemeinsame Benutzermaschine	57
Gliederungsschema der Spezifikation	58
Graphikauswerteprogramm	65
Guarded Command	93
Guarded Region	94
Interne Funktion	57
Interne Operation	57
Internknoten	56
Kommunikation mit technischem Prozeß	2
Kommunikation über Botschaften	1
Kommunikation von Prozessen	2

Kommunikationsknoten	56
Kommunikationsmaschine	55
Kommunikationspartner	58
Kommunikationsstruktur	53, Anhang 2-1
Kontrollflusskonstrukt	19, 29
Korrektheit von Programmen	44
Liste	69
Managementobjekt	25
Maske	1, 63, 73, Anhang 3
Maskensteuerprogramm	67, 97
Nebenläufige Aktion	31
Nichtdeterministische Kontrollanweisungen	1
PASS	1, 53
PEARL	59
PEARL-Modul	29
PEARL-Programm	29
PEARL-Programmbaum	78
PEARL-Programmgerüst	68
PEARL-Task	29
Phasenmodell	39
Private Benutzermaschine	57
Produktverwaltung	26
Programmentwicklung	100
Programmierungsumgebung	1, 6, 59
Programmsynthese	1, 44, 64, 87
Programmteile	87
Programmtransformation	43
Projektbibliothek	6
Projektplanung	25
Prozeßbündel	53
Prozeßgruppe	53
Prozeßsteuerungssoftware	39
Prozeßsystem	67, 74
Prozeßsystem, Definition	54
Regelanwendung	36
Regelauswerter	36
Regelbasis	36
Rohmasken	67
Schnittstelle, graphischer/nichtgraphischer Teil	80

Schnittstelle, Spezifikation/Synthese	81
Semantische Aktion	88, Anhang 1-2,
	Anhang 1-9
Software-Entwicklungsumgebung	3, 36
Software-Lebenszyklus	5, 59
Software-Produktionsumgebung	4
Strukturierungseinheit	53
Taskkopf	90
Taskrumpf	91
Transformationsexpertensystem	36
Transformationsregeln	45
Transformationssystem	36
TREX	36
Umsetzregel	68, 76
Umsetzung	29
Verteiltes PEARL	1, 59
Vordefinierte Symbole	Anhang 2
Wartebereich	55, 90
Werkzeuge	5, 7, 40
Zeichenprogramm	1, 61

Lebenslauf

Am 16. März 1956 wurde ich, Gudrun Kragl, als 2. Kind der Eheleute Reinhold Kragl und Katharina Kragl, geb. Bauer, in Freihung/Opf. geboren.

Von 1962 - 1967 besuchte ich die Volksschule in Freihung, von 1967 - 1973 das Gymnasium in Gemünden am Main. Danach wechselte ich an das Augustinus-Gymnasium in Weiden, im Juni 1976 erlangte ich dort die Allgemeine Hochschulreife.

Im Wintersemester 1976 begann ich an der Universität Erlangen-Nürnberg das Studium der Informatik, im Sommersemester 1982 legte ich die Diplom-Hauptprüfung ab.

Von August 1982 - April 1983 war ich wissenschaftliche Mitarbeiterin im DFG-Projekt Ho 824/2 "Eine Programmiertechnik für Verteilte Systeme von Mikroprozessoren". Seit April 1983 bin ich als wissenschaftliche Hilfskraft am IMMD II der Universität Erlangen-Nürnberg beschäftigt.

Ich besitze die deutsche Staatsangehörigkeit.

In letzter Zeit sind folgende Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung erschienen:

=====

Band 14
=====

- Nr. 1 Pattern Recognition
 (Research at Lehrstuhl für Informatik 5)
 (Februar 1981)

- Nr. 2 Bunke, H.:
 Analyse elektrischer Schaltpläne
 (März 1981)

- Nr. 3 Linster, C.-U.:
 SYMPOS/UNIX - Ein Betriebssystem für homogene
 Polyprozessorsysteme
 (Juni 1981)

- Nr. 4 Akyildiz, J.F., Bolch, G.:
 Analytic Solution Techniques for Queueing Network
 Models of Computer Systems
 (Juli 1981)

- Nr. 5 Lindstedt, W.G.:
 Ein Verfahren zur Erstellung portabler algorithmischer
 Laufzeitprogramme
 (Juli 1981)

- Nr. 6 Vollmar, R.:
 Zum Begriff des Parallelismus bei Polyautomaten
 (Juli 1981)

- Nr. 7 Luft, A.L.:
 Rationaler Sprachgebrauch und orthosprachliche
 Standardisierung als Grundlagen zuverlässiger Software-
 Entwicklung und -Dokumentation
 (Januar 1982)

- Nr. 8 Händler, W.:
 PROCEEDINGS OF A WORKSHOP ON TAXONOMY IN COMPUTER
 ARCHITECTURE
 (Februar 1982)

- Nr. 9 Beth, Th., Hess, P., Wirl, K.:
 Materialien zur Kryptographie
 (März 1982)

- Nr. 10 Jahresbericht 1981 der Informatik
 (März 1982)

Band 15
=====

- Nr. 1 Wittmann, A.:
Ein Mechanismus für die Synchronisation paralleler
Prozesse
(Februar 1982)
- Nr. 2 Maehle, E.:
Fehlertolerantes Verhalten in Multiprozessoren -
Untersuchungen zur Diagnose und Rekonfiguration
(März 1982)
- Nr. 3 Bathelt, P.:
Vergleich von Synchronisationsmechanismen
(Mai 1982)
- Nr. 4 Keramidis, S.:
Eine Methode zur Spezifikation und korrekten
Implementierung von asynchronen Systemen
(Juni 1982)
- Nr. 5 Fromm, H.J.:
Multiprozessor-Rechenanlagen:
Programmstrukturen, Maschinenstrukturen und Zuordnungs-
probleme
(Mai 1982)
- Nr. 6 Bley, H.:
Vorverarbeitung und Segmentierung von Stromlaufplänen
unter Verwendung von Bildgraphen
(Juli 1982)
- Nr. 7 Hchl, W.:
Informatik-Sammlung
Katalog mit historischen Erläuterungen
(August 1982)
- Nr. 8 Messerer, M.:
Ein neuer Ansatz zur Parallelisierung von
Compilern
(August 1982)
- Nr. 9 Grosch, J.:
Eine Programmiersprache mit mengentheoretischen
Konstrukten und deren effiziente Implementierung
(August 1982)
- Nr. 10 Kleinöder, W.:
Stochastische Bewertung von Aufgabenstrukturen
für hierarchische Mehrrechnersysteme
(August 1982)

Band 15 Fortsetzung

=====

- Nr. 11 Kneißl, F.:
Realisierung von Makro-Datenflußmechanismen auf
hierarchischen Mehrrechnersystemen
(August 1982)
- Nr. 12 Brendel, W., Fan, Z., Klar, R., Schmielau, W.:
ERES 82
Handbuch und Fallstudie
Handbook and Case Study
(Oktober 1982)
- Nr. 13 Pelz, K.:
Portable Codegenerierung in Übersetzern
für Prozeß-Programmiersprachen
(Oktober 1982)
- Nr. 14 Sommer, U., Steinchen, J.:
Ein Kleinrechner-Grafiksystem zur Dokumentation
von EDV-Abläufen
(Oktober 1982)
- Nr. 15 Hein, H.-W.:
Das Erlanger Spracherkennungssystem
(November 1982)
- Nr. 16 Bolch, G., Bruchner, W.:
Analytische Modelle für symmetrische Mehrprozessor-
anlagen mit dynamischen Prioritäten
(Dezember 1982)
- Nr. 17 Jahresbericht 1982 der Informatik
(März 1983)

Band 16

=====

- Nr. 1 Gall, R.:
Formale Beschreibung des inkrementellen Programmierens-
im-Großen mit Graph-Grammatiken
(März 1983)
- Nr. 2 König, R.:
Beiträge zur algebraischen Theorie der formalen Sprachen
(März 1983)
- Nr. 3 Welbourn, D.B.:
The Design of Mechanical Components and the Development
of DUCT:
17 Years of CAD/CAM in Cambridge University
(April 1983)
- Nr. 4 Woitok, R.:
Die System-Implementierungs-Sprache ALICE
(Mai 1983)
- Nr. 5 Schneider, S., Schwab, P., Renninger, W.:
Wesen, Vergleich und Stand von Software zur Produktion
von Systemen der computergestützten Unternehmensplanung
(Juli 1983)
- Nr. 6 Händler, W., Herzog, U., Hofmann, F., Schneider, H.J.
und Mitarbeiter:
Projektabschlußbericht 1978 - 1982 Erlangen General Purpose
Array
(Juli 1983)
- Nr. 7 Mackert, L.:
Modellierung, Spezifikation und korrekte Realisierung
von asynchronen Systemen
(Juli 1983)
- Nr. 8 Reitenspieß, M.:
Sprachkonstrukte zur Spezifikation und korrekten Implemen-
tierung von Schutzproblemen
(Juli 1983)
- Nr. 9 Weber, K.:
Modellierung von Fehlverhalten mit Berücksichtigung
paralleler Abläufe
(August 1983)
- Nr. 10 Dietsch, H.:
Ein verteiltes Kommunikationssystem für einen Verbund
aus lokalen Mikrorechner-Netzwerken
(Dezember 1983)

Band 16 Fortsetzung

=====

- Nr. 11 Maehle, E., Schmitter, E.:
Workshop Fehlertolerante Mehrprozessor- und Mehr-
rechnersysteme
(Dezember 1983)
- Nr. 12 Volkert, J.:
Multiplikative Abstandsfunktionen und deren Anwendung
in der Bildverarbeitung
(Dezember 1983)
- Nr. 13 Steinbauer, D.:
Transaktionen als Grundlage zur Strukturierung und
Integritätssicherung in Datenbank-Anwendungssystemen
(Dezember 1983)
- Nr. 14 Wilke, P.:
Zusammenhänge und Unterschiede zwischen Graph-Grammatiken
und Petri-Netzen sowie verwandter Systeme
(Dezember 1983)
- Nr. 15 Jahresbericht der Informatik 1983
(März 1984)

Band 17

=====

- Nr. 1 Eberlein, W.:
Architektur technischer Datenbanken für
integrierte Ingenieursysteme
(Februar 1984)
- Nr. 2 Mertens, P., Heigl, M.:
Neuere Entwicklungen der computergestützten
Produktionsplanung, Eignung - Verbindungen -
Entwicklungspfade
(April 1984)
- Nr. 3 Stoyan, H.:
Maschinen- unabhängige Code-Erzeugung als semantik-
erhaltende beweisbare Programmtransformation
(Juli 1984)
- Nr. 4 Brendel, W.:
Funktionaler Entwurf und automatische Synthese
hochintegrierter Schaltkreise
(August 1984)
- Nr. 5 Brietzmann, A.:
Semantische und pragmatische Analyse im Erlanger
Spracherkennungsprojekt
(Oktober 1984)
- Nr. 6 Hoppe, B.:
Deterministisches Zuordnen vielfach durchlaufener
Aufgabensysteme in Multiprozessorsystemen
(Oktober 1984)
- Nr. 7 Fumy, W., Posch, S., Rieß, H.P.:
Materialien zur Codierungstheorie III
(Dezember 1984)
- Nr. 8 Akyildiz, J. F.:
Leistungsanalyse von Multiprozessorsystemen mit
Prozeßkommunikation
(Dezember 1984)
- Nr. 9 Jahresbericht der Informatik 1984
(März 1985)

- Nr. 1 Siegmann, H.:
Wesen, Vergleich und Stand von zehn ausgewählten
kleinrechnerorientierten Planungssprachen
(Februar 1985)
- Nr. 2 Rathke, M.:
Parallelisieren ordnungserhaltender Programmsysteme
für hierarchische Multiprozessorsysteme
(Juni 1985)
- Nr. 3 Geus, L.:
Parallelisierung eines Mehrgitterverfahrens für die
Navier-Stokes-Gleichungen auf EGPA-Systemen
(Juli 1985)
- Nr. 4 Hellmold, K.-U.:
Der Simulationsrechner SIMPLEX
Eine Multiprozessorkonfiguration zur Simulation
von zeitdiskreten Systemen auf GPSS-FORTRAN-Basis
(September 1985)
- Nr. 5 Klebes, G.:
Entwicklung eines botschaftsorientierten Synchroni-
sationsverfahrens für parallele Prozesse und seine
Realisierung im Rahmen eines Realzeitprogrammiersystems
und
Rupprecht, G.:
Ein Konzept zur Integration von allgemeinen Sprach-
konstrukten zur Synchronisation und von gemeinsamen
abstrakten Datentypen mehrerer Prozesse in Modula-2 und
seine Implementierung
- Nr. 6 Schön, D.:
Technisches Informationssystem und Entwurf integrierter
Ingenieurdatenbanken
(Oktober 1985)
- Nr. 7 Vasconcelos, A.J.V. de:
Berechnung nichtstationärer Vorgänge in Drehstromnetzen
mit Raumzeigerkomponenten auf einem Multiprozessor
(Dezember 1985)
- Nr. 8 Baader, F.:
Die S-Varietät DS und einige Untervarietäten
(Dezember 1985)
- Nr. 9 Jahresbericht der Informatik 1985
(März 1986)

Band 19

=====

- Nr. 1 Erhard, W.:
Feldrechner DAP: Invertierung großer Matrizen
(Februar 1986)
- Nr. 2 Fumy, W.:
Über orthogonale Transformationen und fehlerkorrigierende Codes
(März 1986)
- Nr. 3 Ehrlich, U.:
Ein Lexikon für das natürlich-sprachliche Dialogsystem EVAR
(Mai 1986)
- Nr. 4 Mühlfeld, R.:
Verifikation von Worthypothesen
(Juni 1986)
- Nr. 5 Blank, E., Bolch, G.:
Leistungsbewertung fehlertoleranter Systeme
(Juni 1986)
- Nr. 6 Mertens, P., Allgeyer, K., Däs, H., Schumann, M.:
Betriebliche Expertensysteme in deutschsprachigen Ländern
- Versuch einer Bestandsaufnahme -
(Juli 1986)
- Nr. 7 Moritzen, K.:
Softwarehilfsmittel zur Programmierung von Multiprozessoren mit begrenzten Nachbarschaften
Ein Beitrag zur Konfigurationsverwaltung
(September 1986)
- Nr. 8 Kragl, G.:
Eine Programmierungsumgebung für verteiltes PEARL-Codeerzeugung mittels Programmsynthese
(Oktober 1986)

